Study guide: Introduction to finite element methods

Hans Petter Langtangen[1,2]

Center for Biomedical Computing, Simula Research Laboratory[1]

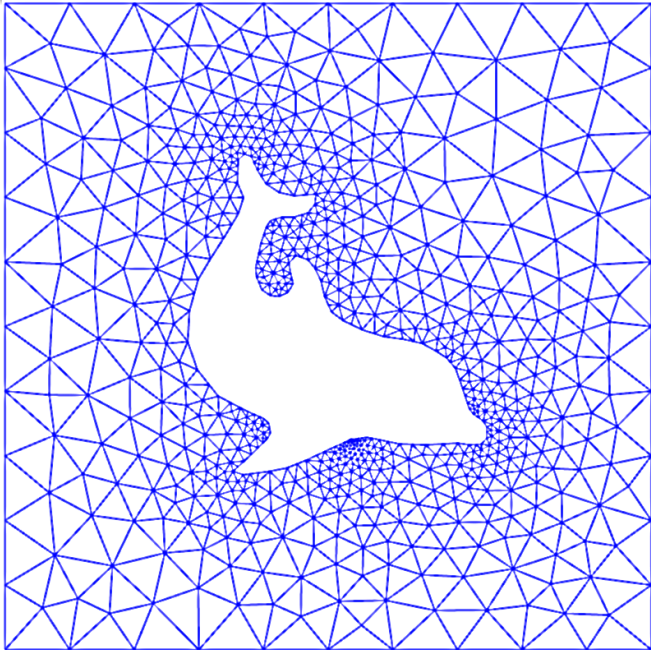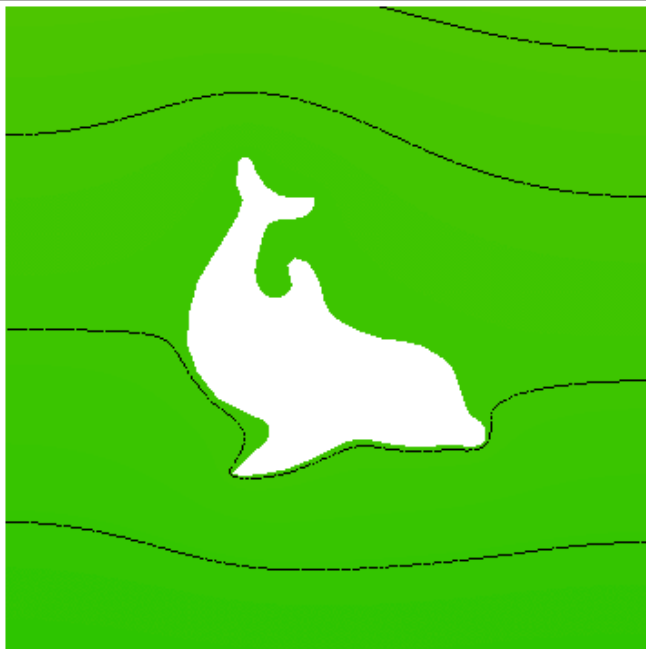Department of Informatics, University of Oslo[2]

Oct 27, 2014

- Can with ease solve PDEs in domains with *complex geometry*
- Can with ease provide higher-order approximations
- Has (in simpler stationary problems) a rigorous mathematical analysis framework (not much considered here)

Solving PDEs by the finite element method

Stationary PDEs:

- Transform the PDE problem to a *variational form*
- Define function approximation over *finite elements*
- Use a machinery to derive *linear systems*
- Solve linear systems

Time-dependent PDEs:

- Finite elements *in space*
- Finite difference (or ODE solver) in time

Learning strategy

- Start with approximation of functions, not PDEs
- Introduce finite element *approximations*
- See later how this machinery is applied to PDEs

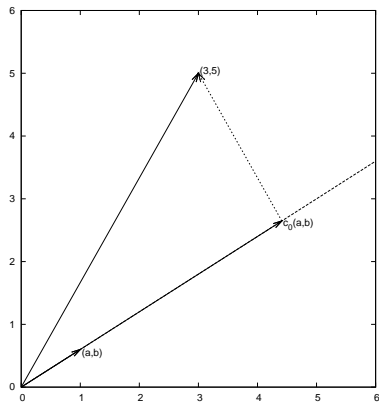**Reason:**

The finite element method has many concepts and a jungle of details. This learning strategy minimizes the mixing of ideas, concepts, and technical details.

General idea of finding an approximation $u(x)$ to some given $f(x)$:

$$u(x) = \sum_{i=0}^{N} c_i \psi_i(x)$$

where

- $\psi_i(x)$ are prescribed functions
- c_i, $i = 0, \ldots, N$ are unknown coefficients to be determined

We shall address three approaches:

- The least squares method
- The projection (or Galerkin) method
- The interpolation (or collocation) method

Underlying motivation for our notation

Our mathematical framework for doing this is phrased in a way such that it becomes easy to understand and use the FEniCS software package for finite element computing.

Given a vector $\boldsymbol{f} = (3, 5)$, find an approximation to $\boldsymbol{f}$ directed along a given line.
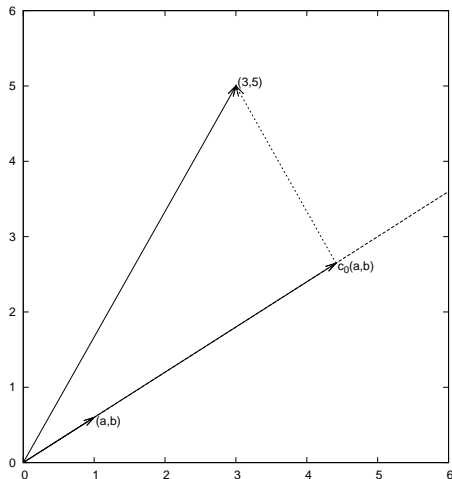
$$V = \text{span}\,\{\psi_0\}$$

- ψ_0 is a basis vector in the space V
- Seek $\boldsymbol{u} = c_0\psi_0 \in V$
- Determine c_0 such that $\boldsymbol{u}$ is the "best" approximation to $\boldsymbol{f}$
- Visually, "best" is obvious

Define

- the error $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$
- the (Eucledian) scalar product of two vectors: $(\boldsymbol{u}, \boldsymbol{v})$
- the norm of $\boldsymbol{e}$: $||\boldsymbol{e}|| = \sqrt{(\boldsymbol{e}, \boldsymbol{e})}$

- Idea: find c_0 such that $||e||$ is minimized
- Mathematical convenience: minimize $E = ||e||^2$

$$\frac{\partial E}{\partial c_0} = 0$$

$$E(c_0) = (e, e) = (f - u, (f - u) = (f - c_0\psi_0, f - c_0\psi_0)$$
$$= (f, f) - 2c_0(f, \psi_0) + c_0^2(\psi_0, \psi_0)$$

$$\frac{\partial E}{\partial c_0} = -2(f, \psi_0) + 2c_0(\psi_0, \psi_0) = 0 \qquad (1)$$

$$c_0 = \frac{(f, \psi_0)}{(\psi_0, \psi_0)} = \frac{3a + 5b}{a^2 + b^2}$$

Observation to be used later: the vanishing derivative (1) can be alternatively written as

$$(e, \psi_0) = 0$$

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for any $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to any $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0$, $\forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0$, $\quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$
  (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0$, $\quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0, \quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0, \quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0, \quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$ (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0, \quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

- Last slide: min E is equivalent with $(e, \psi_0) = 0$
- $(e, \psi_0) = 0$ implies $(e, v) = 0$ for *any* $v \in V$
- That is: instead of using the least-squares principle, we can require that e is orthogonal to *any* $v \in V$
  (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(e, v) = 0, \quad \forall v \in V$
- Equivalent (see notes): find c_0 such that $(e, \psi_0) = 0$
- Insert $e = f - c_0 \psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

Given a vector $\boldsymbol{f}$, find an approximation $\boldsymbol{u} \in V$:

$$V = \text{span}\,\{\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_N\}$$

We have a set of linearly independent basis vectors $\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_N$. Any $\boldsymbol{u} \in V$ can then be written as

$$\boldsymbol{u} = \sum_{j=0}^{N} c_j \boldsymbol{\psi}_j$$

Idea: find $c_0, \ldots, c_N$ such that $E = ||e||^2$ is minimized, $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$.

$$E(c_0, \ldots, c_N) = (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - \sum_j c_j \psi_j, \boldsymbol{f} - \sum_j c_j \psi_j)$$

$$= (\boldsymbol{f}, \boldsymbol{f}) - 2 \sum_{j=0}^{N} c_j (\boldsymbol{f}, \psi_j) + \sum_{p=0}^{N} \sum_{q=0}^{N} c_p c_q (\psi_p, \psi_q)$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0, \ldots, N$$

After some work we end up with a *linear system*

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N \tag{2}$$

$$A_{i,j} = (\psi_i, \psi_j) \tag{3}$$

Can be shown that minimizing $\|e\|$ implies that e is orthogonal to all $v \in V$:

$$(e, v) = 0, \quad \forall v \in V$$

which implies that e most be orthogonal to each basis vector:

$$(e, \psi_i) = 0, \quad i = 0, \ldots, N$$

This orthogonality condition is the principle of the projection (or Galerkin) method. Leads to the same linear system as in the least squares method.

Let V be a *function space* spanned by a set of *basis functions* $\psi_0, \ldots, \psi_N$,

$$V = \text{span}\,\{\psi_0, \ldots, \psi_N\}$$

Find $u \in V$ as a linear combination of the basis functions:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad \mathcal{I}_s = \{0, 1, \ldots, N\}$$

The least squares method can be extended from vectors to functions

As in the vector case, minimize the (square) norm of the error, E, with respect to the coefficients c_j, $j \in \mathcal{I}_s$:

$$E = (e, e) = (f - u, f - u) = \left(f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)\right)$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i =\in \mathcal{I}_s$$

But what is the scalar product when ψ_i is a function?

$$(f, g) = \int_\Omega f(x) g(x) \, dx$$

(natural extension from Eucledian product $(\boldsymbol{u}, \boldsymbol{v}) = \sum_j u_j v_j$)

$$E(c_0, \ldots, c_N) = (e, e) = (f - u, f - u)$$
$$= (f, f) - 2 \sum_{j \in \mathcal{I}_s} c_j(f, \psi_i) + \sum_{p \in \mathcal{I}_s} \sum_{q \in \mathcal{I}_s} c_p c_q(\psi_p, \psi_q)$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = \in \mathcal{I}_s$$

The computations are *identical to the vector case*, and consequently we get a linear system

$$\sum_{j \in \mathcal{I}_s}^{N} A_{i,j} c_j = b_i, \; i \in \mathcal{I}_s, \quad A_{i,j} = (\psi_i, \psi_j), \; b_i = (f, \psi_i)$$

As before, minimizing (e, e) is equivalent to

$$(e, \psi_i) = 0, \quad i \in \mathcal{I}_s$$

which is equivalent to

$$(e, v) = 0, \quad \forall v \in V$$

which is the projection (or Galerkin) method.

The algebra is the same as in the multi-dimensional vector case, and we get the same linear system as arose from the least squares method.

Problem

Approximate a parabola $f(x) = 10(x-1)^2 - 1$ by a straight line.

$$V = \text{span}\,\{1, x\}$$

That is, $\psi_0(x) = 1$, $\psi_1(x) = x$, and $N = 1$. We seek

$$u = c_0\psi_0(x) + c_1\psi_1(x) = c_0 + c_1 x$$

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1 \, dx = 1$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x \, dx = 3/2$$

$$A_{1,0} = A_{0,1} = 3/2$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x \, dx = 7/3$$

$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, dx = 7/3$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, dx = 13/3$$

Solution of 2x2 linear system:

$$c_0 = -38/3, \quad c_1 = 10, \quad u(x) = 10x - \frac{38}{3}$$
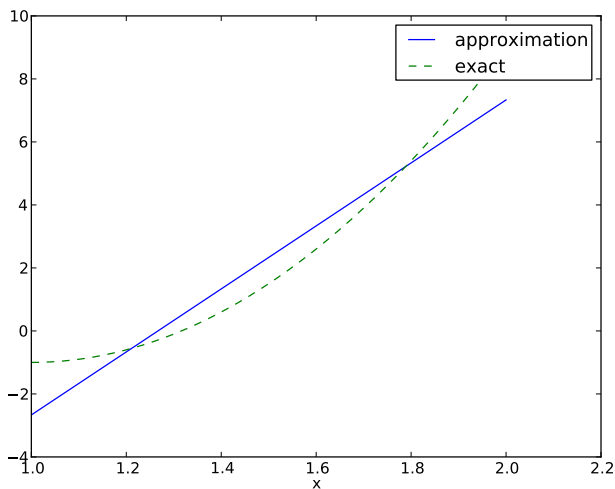
Consider symbolic computation of the linear system, where

- $f(x)$ is given as a `sympy` expression `f` (involving the symbol `x`),
- `psi` is a list of $\{\psi_i\}_{i \in \mathcal{I}_s}$,
- `Omega` is a 2-tuple/list holding the domain Ω

Carry out the integrations, solve the linear system, and return
$u(x) = \sum_j c_j \psi_j(x)$

```python
import sympy as sp

def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sp.integrate(psi[i]*psi[j],
                                  (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sp.integrate(psi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u, c
```

Observe: symmetric coefficient matrix so we can halve the integrations.

- If `sp.integrate` fails, it returns an `sp.Integral` object. We can test on this object and fall back on numerical integration.
- We can include a boolean argument `symbolic` to explicitly choose between symbolic and numerical computing.

```python
def least_squares(f, psi, Omega, symbolic=True):
    ...
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = psi[i]*psi[j]
            if symbolic:
                I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
            if not symbolic or isinstance(I, sp.Integral):
                # Could not integrate symbolically,
                # fall back on numerical integration
                integrand = sp.lambdify([x], integrand)
                I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
            A[i,j] = A[j,i] = I

        integrand = psi[i]*f
        if symbolic:
            I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sp.Integral):
            integrand = sp.lambdify([x], integrand)
```

Compare f and u visually:

```python
def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sp.Symbol('x')
    # Turn f and u to ordinary Python functions
    f = sp.lambdify([x], f, modules="numpy")
    u = sp.lambdify([x], u, modules="numpy")
    resolution = 401  # no of points in plot
    xcoor  = linspace(Omega[0], Omega[1], resolution)
    exact  = f(xcoor)
    approx = u(xcoor)
    plot(xcoor, approx)
    hold('on')
    plot(xcoor, exact)
    legend(['approximation', 'exact'])
    savefig(filename)
```

All code in module `approx1D.py`

```
>>> from approx1D import *
>>> x = sp.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x], Omega=[1, 2])
>>> comparison_plot(f, u, Omega=[1, 2])
```
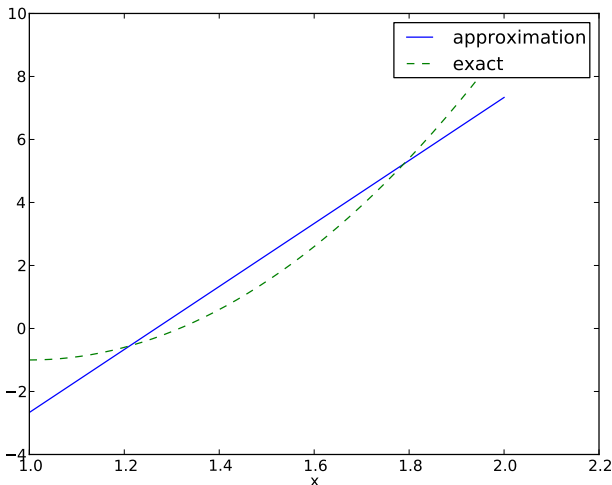
- What if we add $\psi_2 = x^2$ to the space V?
- That is, approximating a parabola by any parabola?
- (Hopefully we get the exact parabola!)

```
>>> from approx1D import *
>>> x = sp.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x, x**2], Omega=[1, 2])
>>> print u
10*x**2 - 20*x + 9
>>> print sp.expand(f)
10*x**2 - 20*x + 9
```

- What if we use $\psi_i(x) = x^i$ for $i = 0, \ldots, N = 40$?
- The output from `least_squares` is $c_i = 0$ for $i > 2$

General result

If $f \in V$, least squares and projection/Galerkin give $u = f$.

If $f \in V$, $f = \sum_{j \in \mathcal{I}_s} d_j \psi_j$, for some $\{d_i\}_{i \in \mathcal{I}_s}$. Then

$$b_i = (f, \psi_i) = \sum_{j \in \mathcal{I}_s} d_j (\psi_j, \psi_i) = \sum_{j \in \mathcal{I}_s} d_j A_{i,j}$$

The linear system $\sum_j A_{i,j} c_j = b_i$, $i \in \mathcal{I}_s$, is then

$$\sum_{j \in \mathcal{I}_s} c_j A_{i,j} = \sum_{j \in \mathcal{I}_s} d_j A_{i,j}, \quad i \in \mathcal{I}_s$$

which implies that $c_i = d_i$ for $i \in \mathcal{I}_s$ and u is identical to f.

The previous computations were symbolic. What if we solve the linear system numerically with standard arrays?

That is, f is parabola, but we approximate with

$$u(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_N x^N$$

We expect $c_2 = c_3 = \cdots = c_N = 0$ since $f \in V$ implies $u = f$.

Will we get this result with finite precision computer arithmetic?

exact	sympy	numpy32	numpy64
9	9.62	5.57	8.98
-20	-23.39	-7.65	-19.93
10	17.74	-4.50	9.96
0	-9.19	4.13	-0.26
0	5.25	2.99	0.72
0	0.18	-1.21	-0.93
0	-2.48	-0.41	0.73
0	1.81	-0.013	-0.36
0	-0.66	0.08	0.11
0	0.12	0.04	-0.02
0	-0.001	-0.02	0.002

- Column 2: `matrix` and `lu_solve` from `sympy.mpmath.fp`
- Column 3: numpy matrix with 4-byte floats
- Column 4: numpy matrix with 8-byte floats

The ill-conditioning is due to almost linearly dependent basis functions for large N

- Significant round-off errors in the numerical computations (!)
- But if we plot the approximations they look good (!)

Source or problem: the x^i functions become almost linearly dependent as i grows:
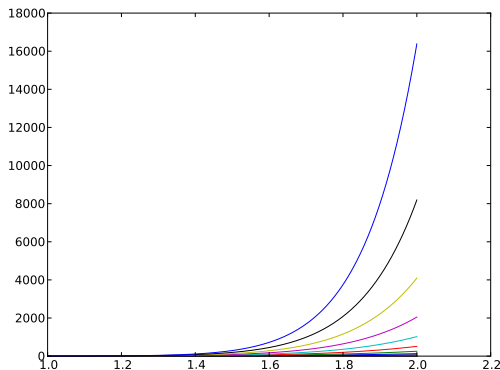
- Almost linearly dependent basis functions give almost singular matrices
- Such matrices are said to be *ill conditioned*, and Gaussian elimination is severely affected by round-off errors
- The basis $1, x, x^2, x^3, x^4, \ldots$ is a bad basis
- Polynomials are fine as basis, but the more orthogonal they are, $(\psi_i, \psi_j) \approx 0$, the better

Let's approximate f by a typical Fourier series expansion

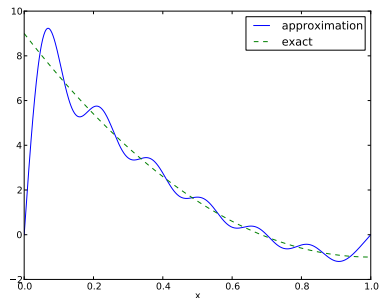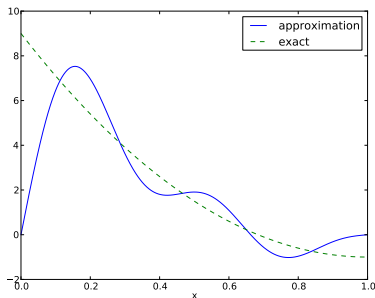$$u(x) = \sum_i a_i \sin i\pi x = \sum_{j=0}^{N} c_j \sin((j+1)\pi x)$$

which means that

$$V = \text{span}\left\{\sin \pi x, \sin 2\pi x, \ldots, \sin(N+1)\pi x\right\}$$

Computations using the `least_squares` function:

```
N = 3
from sympy import sin, pi
psi = [sin(pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
```

Left: $N = 3$, right: $N = 11$:

Problem:

All $\psi_i(0) = 0$ and hence $u(0) = 0 \neq f(0) = 9$. Similar problem at $x = 1$. The boundary values of u are always wrong!
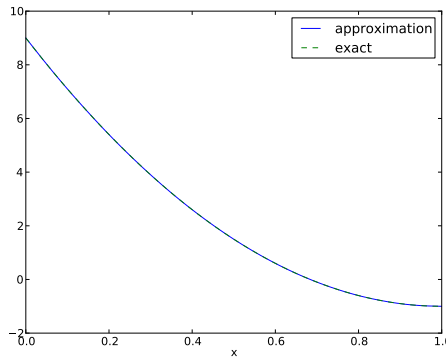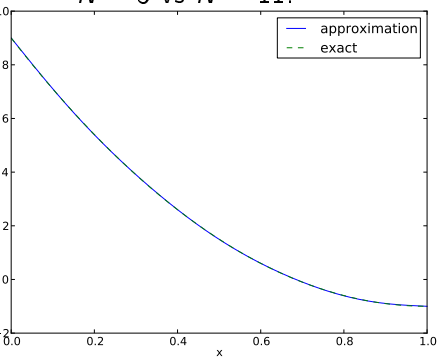
- Considerably improvement with $N = 11$, but still undesired discrepancy at $x = 0$ and $x = 1$
- Possible remedy: add a term that leads to correct boundary values

$$u(x) = f(0)(1 - x) + xf(1) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

The extra terms ensure $u(0) = f(0)$ and $u(1) = f(1)$ and is a strikingly good help to get a good approximation!

$N = 3$ vs $N = 11$:

This choice of sine functions as basis functions is popular because

- the basis functions are orthogonal: $(\psi_i, \psi_j) = 0$
- implying that $A_{i,j}$ is a diagonal matrix
- implying that we can solve for $c_i = 2 \int_0^1 f(x) \sin((i+1)\pi x) dx$
- and what we get is the standard Fourier sine series of f

In general, for an orthogonal basis, $A_{i,j}$ is diagonal and we can easily solve for c_i:

$$c_i = \frac{b_i}{A_{i,i}} = \frac{(f, \psi_i)}{(\psi_i, \psi_i)}$$

Function for the least squares method with orthogonal basis functions

```python
def least_squares_orth(f, psi, Omega):
    N = len(psi) - 1
    A = [0]*(N+1)
    b = [0]*(N+1)
    x = sp.Symbol('x')
    for i in range(N+1):
        A[i] = sp.integrate(psi[i]**2, (x, Omega[0], Omega[1]))
        b[i] = sp.integrate(psi[i]*f,  (x, Omega[0], Omega[1]))
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    for i in range(len(psi)):
        u += c[i]*psi[i]
    return u, c
```

Extensions:

- We can choose between symbolic or numerical integration (symbolic argument).
- If symbolic, we fall back on numerical integration after failure (sp.Integral is returned from sp.integrate).

```python
def least_squares_orth(f, psi, Omega, symbolic=True):
    ...
    for i in range(N+1):
        # Diagonal matrix term
        A[i] = sp.integrate(psi[i]**2, (x, Omega[0], Omega[1]))

        # Right-hand side term
        integrand = psi[i]*f
        if symbolic:
            I = sp.integrate(integrand,  (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sp.Integral):
            print 'numerical integration of', integrand
            integrand = sp.lambdify([x], integrand)
            I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i] = I
```

Here is another idea for approximating $f(x)$ by $u(x) = \sum_j c_j \psi_j$:

- Force $u(x_i) = f(x_i)$ at some selected *collocation* points $\{x_i\}_{i \in \mathcal{I}_s}$
- Then u is said to *interpolate* f
- The method is known as *interpolation* or *collocation*

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = f(x_i) \quad i \in \mathcal{I}_s, N$$

This is a linear system with no need for integration:

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s \tag{5}$$

$$A_{i,j} = \psi_j(x_i) \tag{6}$$

$$b_i = f(x_i) \tag{7}$$

No symmetric matrix: $\psi_j(x_i) \neq \psi_i(x_j)$ in general

points holds the interpolation/collocation points

```python
def interpolation(f, psi, points):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    # Turn psi and f into Python functions
    psi = [sp.lambdify([x], psi[i]) for i in range(N+1)]
    f = sp.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = psi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i](x)
    return u
```

The collocation or interpolation method; approximating a parabola by linear functions

- Potential difficulty: how to choose x_i?
- The results are sensitive to the points!

$(4/3, 5/3)$ vs $(1, 2)$:
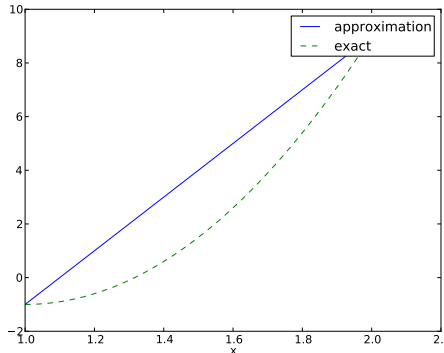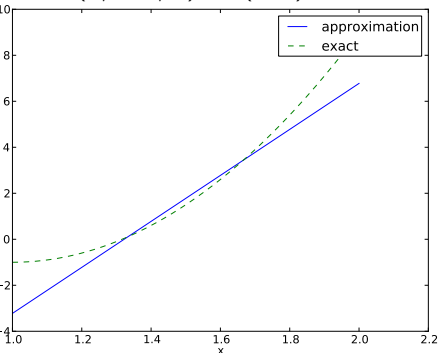
Motivation:

- The interpolation/collocation method avoids integration
- With a diagonal matrix $A_{i,j} = \psi_j(x_i)$ we can solve the linear system by hand

The *Lagrange interpolating polynomials* ψ_j have the property that

$$\psi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \left\{ \begin{array}{ll} 1, & i = j \\ 0, & i \neq j \end{array} \right.$$

Hence, $c_i = f(x_i)$ and

$$u(x) = \sum_{j \in \mathcal{I}_s} f(x_i)\psi_i(x)$$

- Lagrange polynomials and interpolation/collocation look convenient

$$\psi_i(x) = \prod_{j=0, j \neq i}^{N} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_N}{x_i - x_N}$$
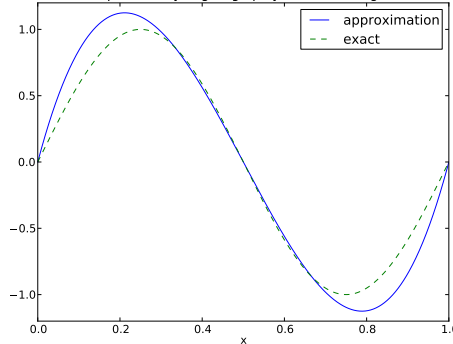
```python
def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```
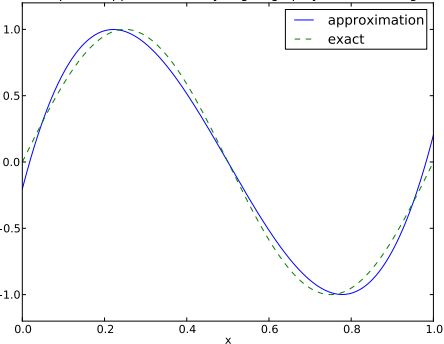
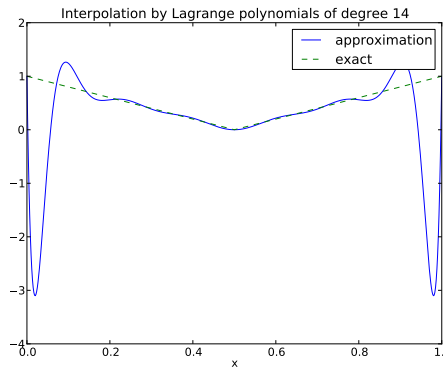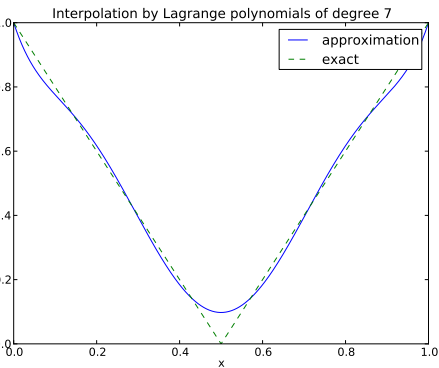Interpolation by Lagrange polynomials of degree 7

Interpolation by Lagrange polynomials of degree 14

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.
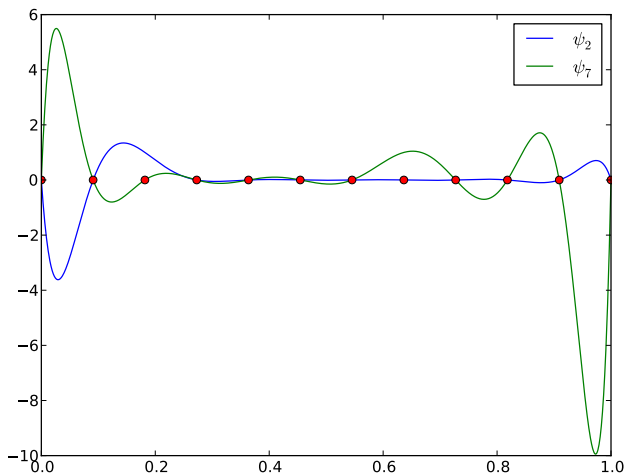
Problem: strong oscillations near the boundaries for larger N values

The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a) \cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0 \ldots, N$$

on an interval $[a, b]$.

Interpolation by Lagrange polynomials of degree 7

Interpolation by Lagrange polynomials of degree 14

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.

In the finite element method we use basis functions with local support

- *Local support*: $\psi_i(x) \neq 0$ for x in a small subdomain of Ω
- Typically hat-shaped
- $u(x)$ based on these ψ_i is a piecewise polynomial defined over many (small) subdomains
- We introduce φ_i as the name of these finite element hat functions (and for now choose $\psi_i = \varphi_i$)
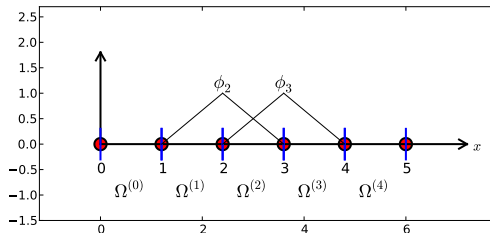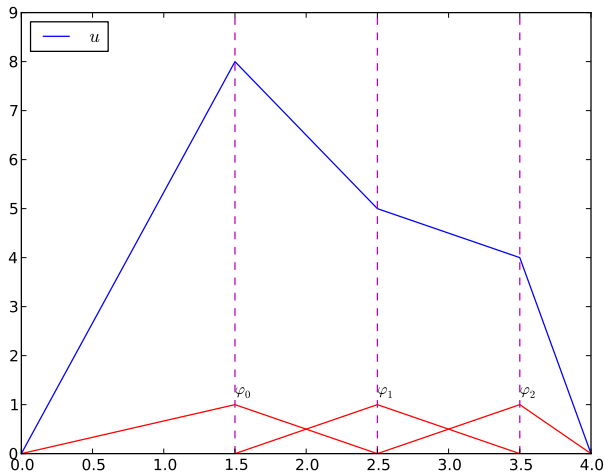
Split Ω into non-overlapping subdomains called *elements*:

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(N_e)}$$

On each element, introduce points called *nodes*: $x_0, \ldots, x_{N_n}$

- The finite element basis functions are named $\varphi_i(x)$
- $\varphi_i = 1$ at node i and 0 at all other nodes
- φ_i is a Lagrange polynomial on each element
- For nodes at the boundary between two elements, φ_i is made up of a Lagrange polynomial over each element
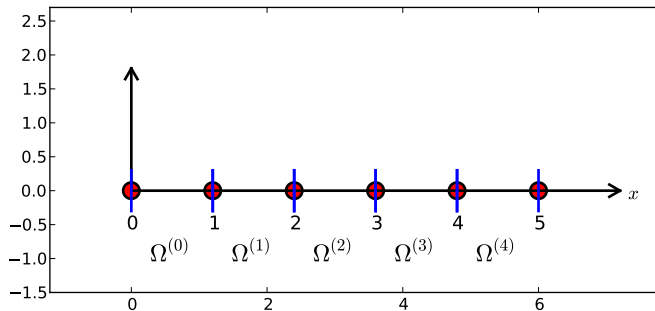
Data structure: `nodes` holds coordinates or nodes, `elements` holds
the node numbers in each element

```
nodes = [0, 1.2, 2.4, 3.6, 4.8, 5]
elements = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]]
```
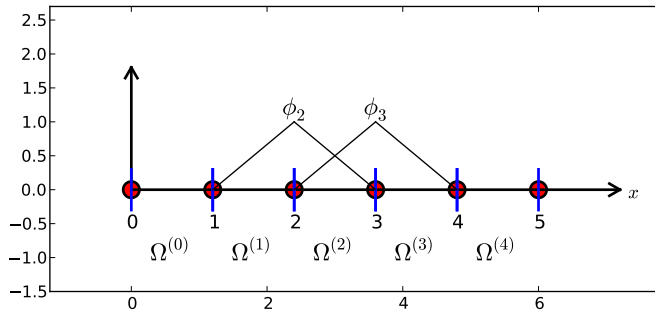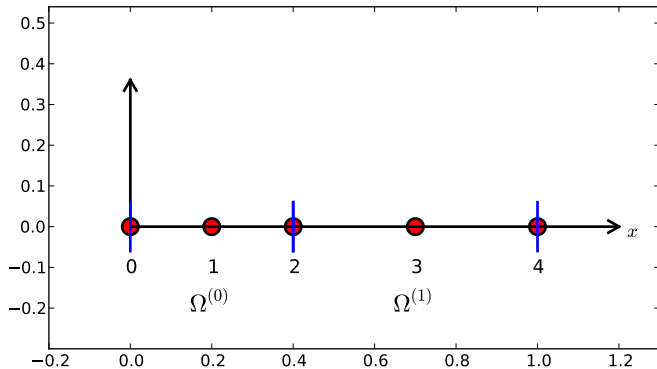
```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```
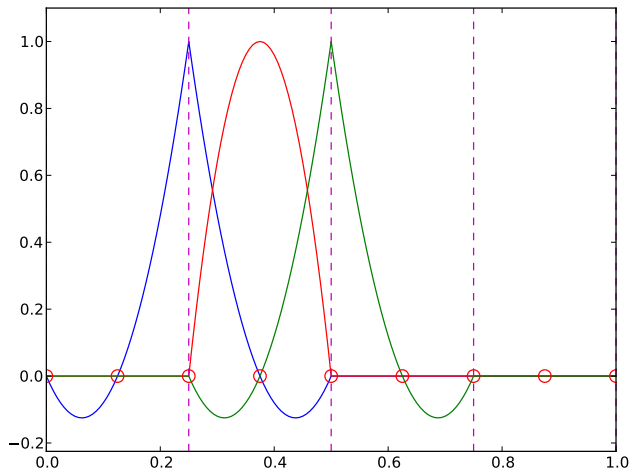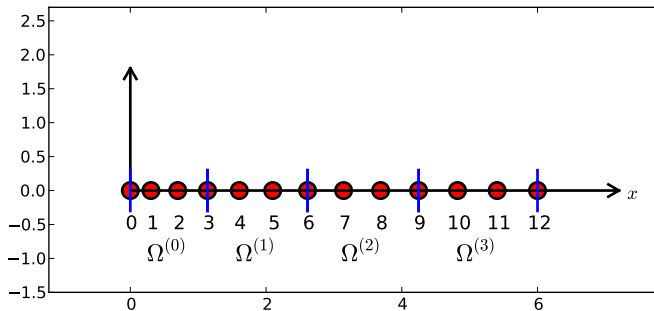
```
d = 3   # d+1 nodes per element
num_elements = 4
num_nodes = num_elements*d + 1
nodes = [i*0.5 for i in range(num_nodes)]
elements = [[i*d+j for j in range(d+1)] for i in range(num_elements)]
```
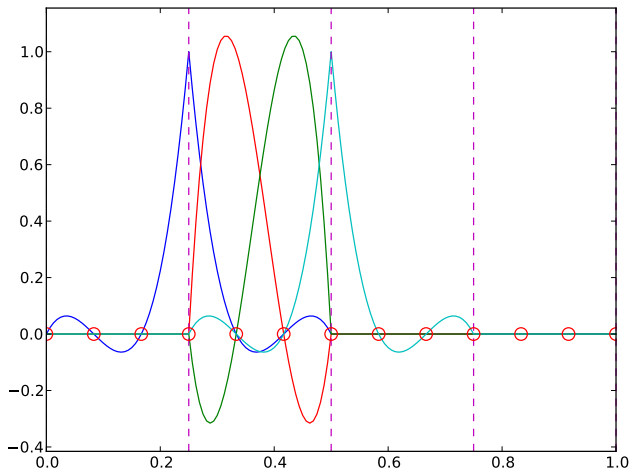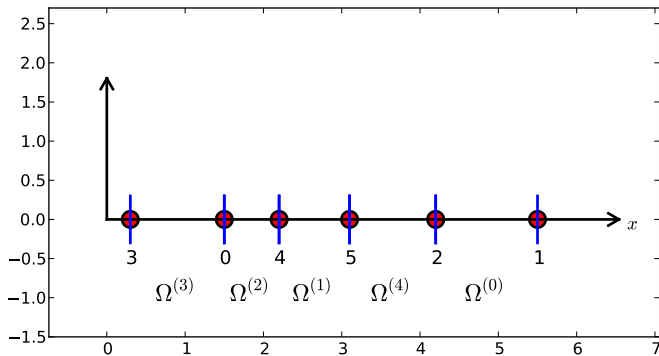
```
nodes = [1.5, 5.5, 4.2, 0.3, 2.2, 3.1]
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

Important property: c_i is the value of u at node i, x_i:

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = c_i \varphi_i(x_i) = c_i$$

because $\varphi_j(x_i) = 0$ if $i \neq j$ and $\varphi_i(x_i) = 1$

- $\varphi_i(x) \neq 0$ only on those elements that contain global node i
- $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if i and j are global node numbers in the same element

Since $A_{i,j} = \int \varphi_i \varphi_j \, \mathrm{d}x$, *most of the elements in the coefficient matrix will be zero*
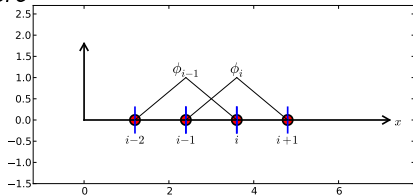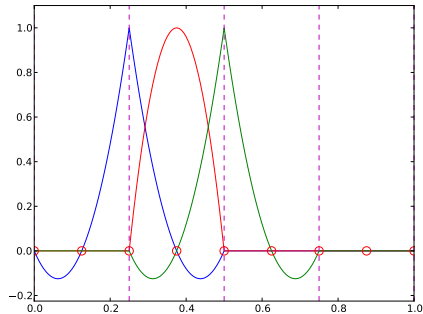
1. Associate Lagrange polynomials with the nodes in an element
2. When the polynomial is 1 on the element boundary, combine it with the polynomial in the neighboring element that is also 1 at the same point
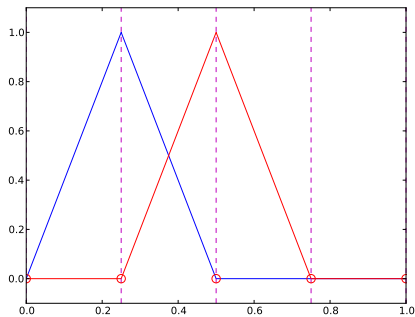
$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1} \\ (x - x_{i-1})/h & x_{i-1} \le x < x_i \\ 1 - (x - x_i)/h, & x_i \le x < x_{i+1} \\ 0, & x \ge x_{i+1} \end{cases}$$

$A_{2,3} = \int_\Omega \varphi_2 \varphi_3 dx$: $\varphi_2 \varphi_3 \neq 0$ only over element 2. There,

$$\varphi_3(x) = (x - x_2)/h, \quad \varphi_2(x) = 1 - (x - x_2)/h$$

$$A_{2,3} = \int_\Omega \varphi_2 \varphi_3 \, dx = \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right) \frac{x - x_2}{h} \, dx = \frac{h}{6}$$
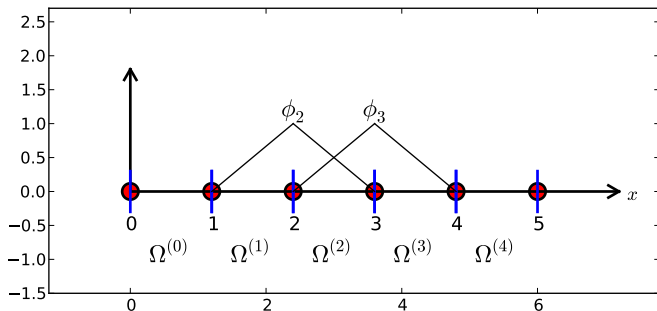
$$A_{2,2} = \int_{x_1}^{x_2} \left(\frac{x - x_1}{h} \right)^2 \, \mathrm{d}x + \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h} \right)^2 \, \mathrm{d}x = \frac{2h}{3}$$
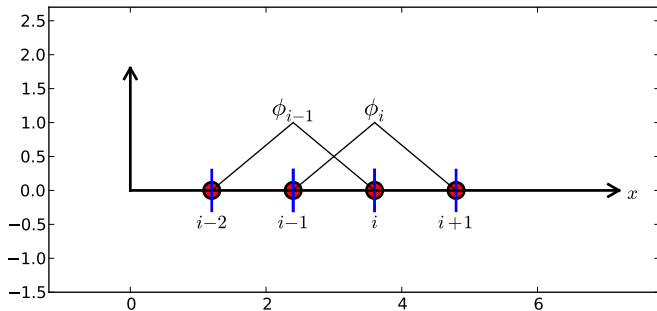
$$A_{i,i-1} = \int_{\Omega} \varphi_i \varphi_{i-1} \, \mathrm{d}x = ?$$

$$A_{i,i-1} = \int_\Omega \varphi_i \varphi_{i-1} \, dx$$

$$= \underbrace{\int_{x_{i-2}}^{x_{i-1}} \varphi_i \varphi_{i-1} \, dx}_{\varphi_i = 0} + \int_{x_{i-1}}^{x_i} \varphi_i \varphi_{i-1} \, dx + \underbrace{\int_{x_i}^{x_{i+1}} \varphi_i \varphi_{i-1} \, dx}_{\varphi_{i-1} = 0}$$

$$= \int_{x_{i-1}}^{x_i} \underbrace{\left(\frac{x - x_i}{h} \right)}_{\varphi_i(x)} \underbrace{\left(1 - \frac{x - x_{i-1}}{h} \right)}_{\varphi_{i-1}(x)} dx = \frac{h}{6}$$

- $A_{i,i+1} = A_{i,i-1}$ due to symmetry
- $A_{i,i} = 2h/3$ (same calculation as for $A_{2,2}$)
- $A_{0,0} = A_{N,N} = h/3$ (only one element)
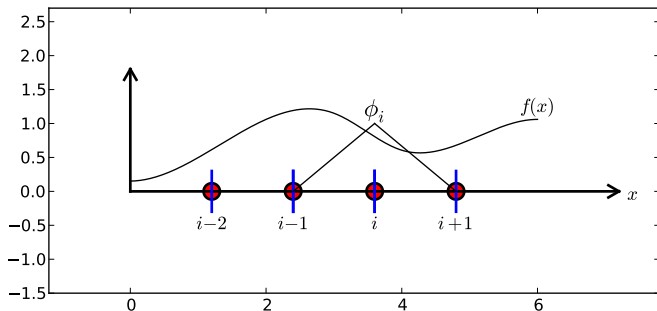
$$b_i = \int_\Omega \varphi_i(x) f(x) \, dx = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x) \, dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h} \right) f(x)$$

Need a specific $f(x)$ to do more...

- $f(x) = x(1 - x)$ on $\Omega = [0, 1]$
- Two equal-sized elements $[0, 0.5]$ and $[0.5, 1]$

$$A = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \quad b = \frac{h^2}{12} \begin{pmatrix} 2 - 3h \\ 12 - 14h \\ 10 - 17h \end{pmatrix}$$

$$c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2$$
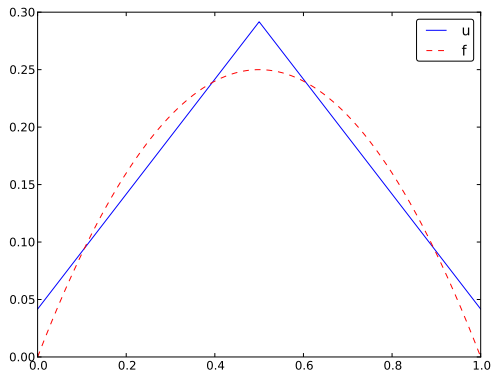
$$u(x) = c_0 \varphi_0(x) + c_1 \varphi_1(x) + c_2 \varphi_2(x)$$
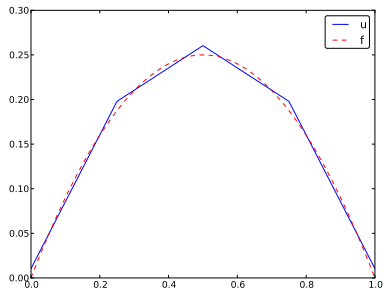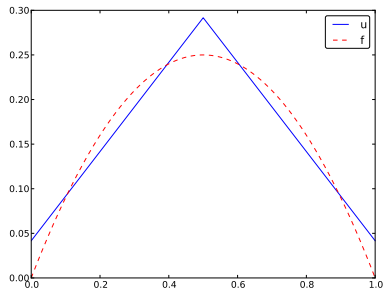
Recall: if $f \in V$, u becomes exact. When f is a parabola, any choice of P2 elements (1 or many) will give $u = f$ exactly. The same is true for P3, P4, ... elements since all of them can represent a 2nd-degree polynomial exactly.
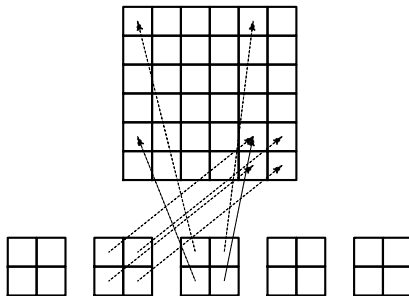
$$A_{i,j} = \int_\Omega \varphi_i \varphi_j dx = \sum_e \int_{\Omega^{(e)}} \varphi_i \varphi_j dx, \quad A_{i,j}^{(e)} = \int_{\Omega^{(e)}} \varphi_i \varphi_j dx$$

Important observations:

- $A_{i,j}^{(e)} \neq 0$ if and only if i and j are nodes in element e (otherwise no overlap between the basis functions)
- All the nonzero elements in $A_{i,j}^{(e)}$ are collected in an *element matrix*
- The element matrix has contributions from the φ_i functions associated with the nodes in element
- It is convenient to introduce a *local numbering* of the nodes in an element: $0, 1, \ldots, d$
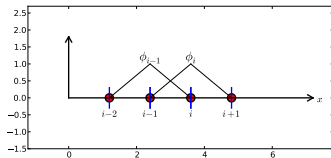
$$\tilde{A}^{(e)} = \{\tilde{A}^{(e)}_{r,s}\}, \quad \tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)} \varphi_{q(e,s)} dx, \quad r,s \in I_d = \{0, \ldots, d\}$$

Now,

- r, s run over *local node numbers* in an element: $0, 1, \ldots, d$
- i, j run over *global node numbers* $i, j \in \mathcal{I}_s = \{0, 1, \ldots, N\}$
- $i = q(e, r)$: mapping of local node number r in element e to the global node number i (math equivalent to `i=elements[e][r]`)
- Add $\tilde{A}^{(e)}_{r,s}$ into the global $A_{i,j}$ (*assembly*)

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad r,s \in I_d$$
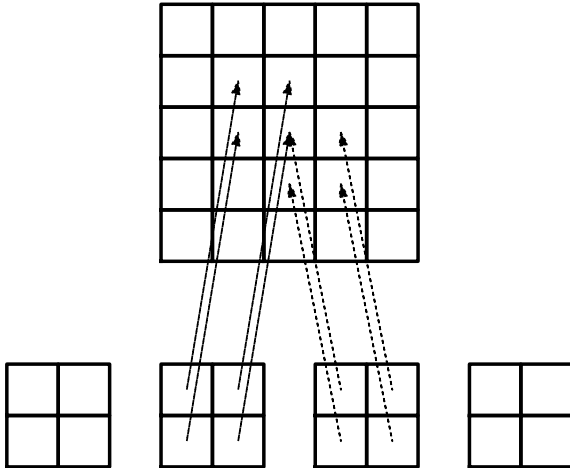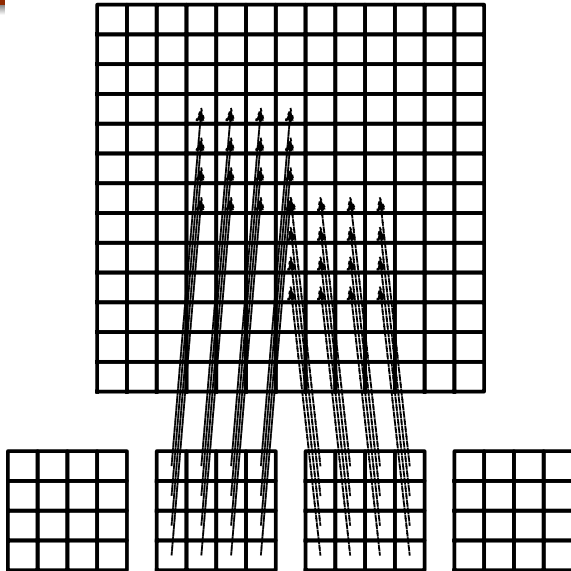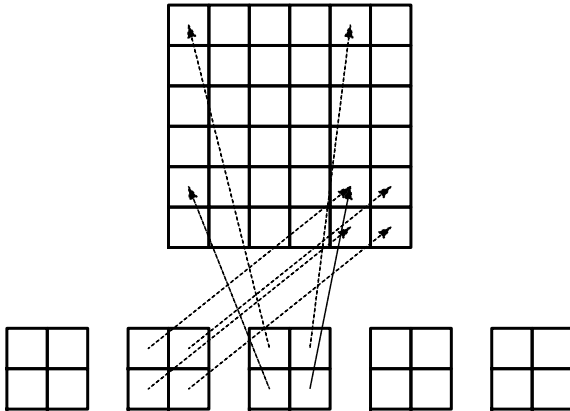
Animation

Animation

Animation

$$b_i = \int_\Omega f(x)\varphi_i(x)dx = \sum_e \int_{\Omega^{(e)}} f(x)\varphi_i(x)dx, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x)\varphi_i(x)dx$$
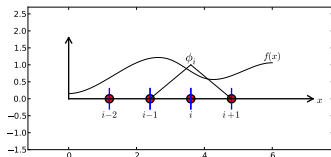
Important observations:

- $b_i^{(e)} \neq 0$ if and only if global node i is a node in element e (otherwise $\varphi_i = 0$)
- The $d + 1$ nonzero $b_i^{(e)}$ can be collected in an *element vector* $\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}$, $r \in I_d$

Assembly:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s \in I_d$$

Instead of computing

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{x_L}^{x_R} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx$$

we now map $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$ with local coordinate X

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X$$

or rewritten as

$$x = x_m + \frac{1}{2}hX, \qquad x_m = (x_L + x_R)/2, \quad h = x_R - x_L$$

Reference element integration: just change integration variable from x to X. Introduce local basis function

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X))$$

$$\tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X) \underbrace{\frac{dx}{dX}}_{\det J = h/2} dX = \int_{-1}^{1} \tilde{\varphi}_r($$

$$\tilde{b}^{(e)}_r = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J \, dX$$

- Always the same domain for integration: $[-1, 1]$
- We only need formulas for $\tilde{\varphi}_r(X)$ over one element (no *piecewise* polynomial definition)
- $\tilde{\varphi}_r(X)$ is the same for all elements: no dependence on element length and location, which is "factored out" in the mapping and $\det J$

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{8}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{9}$$

Note: simple polynomial expressions (no need to consider piecewisely defined functions)

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \qquad (10)$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \qquad (11)$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \qquad (12)$$

Easy to generalize to arbitrary order!

Three alternatives:

1. Map the global basis function $\varphi_i(x)$ over an element to X coordinates
2. Compute $\tilde{\varphi}_r(X)$ from scratch using
   1. a given polynomial order d
   2. $\tilde{\varphi}_r(X) = 1$ at local node 1
   3. $\tilde{\varphi}_r(X) = 1$ at all other local nodes
3. Use formulas for Lagrange interpolating polynomials on the element

P1 elements and $f(x) = x(1 - x)$.

$$\tilde{A}_{0,0}^{(e)} = \int_{-1}^{1} \tilde{\varphi}_0(X)\tilde{\varphi}_0(X)\frac{h}{2}dX$$

$$= \int_{-1}^{1} \frac{1}{2}(1 - X)\frac{1}{2}(1 - X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1 - X)^2 dX = \frac{h}{3}$$

$$(13)$$

$$\tilde{A}_{1,0}^{(e)} = \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_0(X)\frac{h}{2}dX$$

$$= \int_{-1}^{1} \frac{1}{2}(1 + X)\frac{1}{2}(1 - X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1 - X^2)dX = \frac{h}{6}$$

$$(14)$$

$$\tilde{A}_{0,1}^{(e)} = \tilde{A}_{1,0}^{(e)} \tag{15}$$

$$\tilde{A}_{1,1}^{(e)} = \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_1(X)\frac{h}{2}dX$$

$$\tilde{b}_0^{(e)} = \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX$$

$$= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1 - X)\frac{h}{2}dX$$

$$= -\frac{1}{24}h^3 + \frac{1}{6}h^2x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m \qquad (17)$$

$$\tilde{b}_1^{(e)} = \int_{-1}^{1} f(x(X))\tilde{\varphi}_1(X)\frac{h}{2}dX$$

$$= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1 + X)\frac{h}{2}dX$$

$$= -\frac{1}{24}h^3 - \frac{1}{6}h^2x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m \qquad (18)$$

x_m: element midpoint.

```
>>> import sympy as sp
>>> x, x_m, h, X = sp.symbols('x x_m h X')
>>> sp.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sp.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
>>> b_0 = sp.integrate(h/4*x*(1-x)*(1-X), (X, -1, 1))
>>> print b_0
-h**3/24 + h**2*x_m/6 - h**2/12 - h*x_m**2/2 + h*x_m/2
```

Can print out in LaTeX too (convenient for copying into reports):

```
>>> print sp.latex(b_0, mode='plain')
- \frac{1}{24} h^{3} + \frac{1}{6} h^{2} x_{m}
- \frac{1}{12} h^{2} - \half h x_{m}^{2}
+ \half h x_{m}
```

- Coming functions appear in `fe_approx1D.py`
- Functions can operate in symbolic or numeric mode
- The code documents all steps in finite element calculations!

Compute finite element basis functions in the reference element

Let $\tilde{\varphi}_r(X)$ be a Lagrange polynomial of degree d:

```python
import sympy as sp
import numpy as np

def phi_r(r, X, d):
    if isinstance(X, sp.Symbol):
        h = sp.Rational(1, d)  # node spacing
        nodes = [2*i*h - 1 for i in range(d+1)]
    else:
        # assume X is numeric: use floats for nodes
        nodes = np.linspace(-1, 1, d+1)
    return Lagrange_polynomial(X, r, nodes)

def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p

def basis(d=1):
    """Return the complete basis."""
    X = sp.Symbol('X')
    phi = [phi_r(r, X, d) for r in range(d+1)]
    return phi
```

```python
def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sp.zeros((n, n))
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    detJ = h/2   # dx/dX
    for r in range(n):
        for s in range(r, n):
            A_e[r,s] = sp.integrate(phi[r]*phi[s]*detJ, (X, -1, 1))
            A_e[s,r] = A_e[r,s]
    return A_e
```

```
>>> from fe_approx1D import *
>>> phi = basis(d=1)
>>> phi
[1/2 - X/2, 1/2 + X/2]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=True)
[h/3, h/6]
[h/6, h/3]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=False)
[0.0333333333333333, 0.0166666666666667]
[0.0166666666666667, 0.0333333333333333]
```

```python
def element_vector(f, phi, Omega_e, symbolic=True):
    n = len(phi)
    b_e = sp.zeros((n, 1))
    # Make f a function of X
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    x = (Omega_e[0] + Omega_e[1])/2 + h/2*X   # mapping
    f = f.subs('x', x)   # substitute mapping formula for x
    detJ = h/2   # dx/dX
    for r in range(n):
        b_e[r] = sp.integrate(f*phi[r]*detJ, (X, -1, 1))
    return b_e
```

Note f.subs('x', x): replace x by $x(X)$ such that f contains X

- Element matrix: only polynomials and `sympy` always succeeds
- Element vector: $\int f\tilde{\varphi}\,\mathrm{d}x$ can fail (`sympy` then returns an `Integral` object instead of a number)

```
def element_vector(f, phi, Omega_e, symbolic=True):
        ...
        I = sp.integrate(f*phi[r]*detJ, (X, -1, 1))  # try...
        if isinstance(I, sp.Integral):
            h = Omega_e[1] - Omega_e[0]   # Ensure h is numerical
            detJ = h/2
            integrand = sp.lambdify([X], f*phi[r]*detJ)
            I = sp.mpmath.quad(integrand, [-1, 1])
        b_e[r] = I
        ...
```

```python
def assemble(nodes, elements, phi, f, symbolic=True):
    N_n, N_e = len(nodes), len(elements)
    zeros = sp.zeros if symbolic else np.zeros
    A = zeros((N_n, N_n))
    b = zeros((N_n, 1))
    for e in range(N_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

        A_e = element_matrix(phi, Omega_e, symbolic)
        b_e = element_vector(f, phi, Omega_e, symbolic)

        for r in range(len(elements[e])):
            for s in range(len(elements[e])):
                A[elements[e][r],elements[e][s]] += A_e[r,s]
            b[elements[e][r]] += b_e[r]
    return A, b
```

```
if symbolic:
    c = A.LUsolve(b)            # sympy arrays, symbolic Gaussian elim.
else:
    c = np.linalg.solve(A, b)   # numpy arrays, numerical solve
```

Note: the symbolic computation of A, b and A.LUsolve(b) can be very tedious.

```
>>> h, x = sp.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,    0]
[h/6, 2*h/3, h/6]
[  0,   h/6, h/3]
>>> b
[     h**2/6 - h**3/12]
[      h**2 - 7*h**3/6]
[5*h**2/6 - 17*h**3/12]
>>> c = A.LUsolve(b)
>>> c
[                        h**2/6]
[12*(7*h**2/12 - 35*h**3/72)/(7*h)]
[  7*(4*h**2/7 - 23*h**3/21)/(2*h)]
```

```
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> x = sp.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.166666666666667, 0.0833333333333333,                 0]
[0.0833333333333333,  0.333333333333333, 0.0833333333333333]
[                 0, 0.0833333333333333,  0.166666666666667]
>>> b
[         0.03125]
[0.104166666666667]
[         0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.0416666666666666]
[ 0.291666666666667]
[0.0416666666666666]
```

```
>>> d=1; N_e=8; Omega=[0,1]   # 8 linear elements on [0,1]
>>> phi = basis(d)
>>> f = x*(1-x)
>>> nodes, elements = mesh_symbolic(N_e, d, Omega)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,     0,      0,      0,      0,      0,      0,     0]
[h/6, 2*h/3,   h/6,      0,      0,      0,      0,      0,     0]
[  0,   h/6, 2*h/3,    h/6,      0,      0,      0,      0,     0]
[  0,     0,   h/6,  2*h/3,    h/6,      0,      0,      0,     0]
[  0,     0,     0,    h/6,  2*h/3,    h/6,      0,      0,     0]
[  0,     0,     0,      0,    h/6,  2*h/3,    h/6,      0,     0]
[  0,     0,     0,      0,      0,    h/6,  2*h/3,    h/6,     0]
[  0,     0,     0,      0,      0,      0,    h/6,  2*h/3,  h/6]
[  0,     0,     0,      0,      0,      0,      0,    h/6,   h/3]
```

Note: do this by hand to understand what is going on!

- Sparse = most of the entries are zeros
- Below: P1 elements

$$
A = \frac{h}{6}
\begin{pmatrix}
2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
1 & 4 & 1 & \ddots & & & & & \vdots \\
0 & 1 & 4 & 1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & 1 & 4 & 1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2
\end{pmatrix}
$$

$$A = \frac{h}{30}
\begin{pmatrix}
4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4
\end{pmatrix}$$
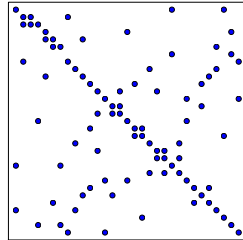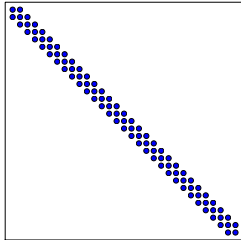
- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily

- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily
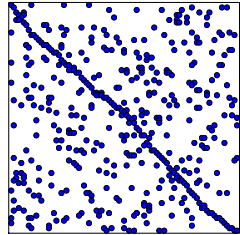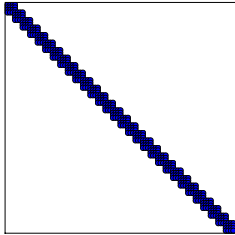
The minimum storage requirements for the coefficient matrix $A_{i,j}$:

- P1 elements: only 3 nonzero entries per row
- P2 elements: only 5 nonzero entries per row
- P3 elements: only 7 nonzero entries per row
- It is important to utilize sparse storage and sparse solvers
- In Python: `scipy.sparse` package

Compute a mesh with N_e elements, basis functions of degree d, and approximate a given symbolic expression $f(x)$ by a finite element expansion $u(x) = \sum_j c_j \varphi_j(x)$:

```python
import sympy as sp
from fe_approx1D import approximate
x = sp.Symbol('x')

approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=4)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=2)
approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=8)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=4)
```

- Finite difference approximation u_i of a function $f(x)$: simply choose $u_i = f(x_i)$
- This is the same as $u \approx \sum_i c_i \varphi_i$ + interpolation (see next slide)
- $u \approx \sum_i c_i \varphi_i$ + Galerkin/projection or least squares method: must derive and solve a linear system
- What is *really* the difference in the approximation u?

Let $\{x_i\}_{i\in\mathcal{I}_s}$ be the nodes in the mesh. Collocation/interpolation means

$$u(x_i) = f(x_i), \quad i \in \mathcal{I}_s,$$

which translates to

$$\sum_{j\in\mathcal{I}_s} c_j \varphi_j(x_i) = f(x_i),$$

but $\varphi_j(x_i) = 0$ if $i \neq j$ so the sum collapses to one term $c_i \varphi_i(x_i) = c_i$, and we have the result

$$c_i = f(x_i)$$

Same result as the standard finite difference approach, but finite elements define u also *between* the x_i points

- Scope: work with P1 elements
- Use projection/Galerkin or least squares (equivalent)
- Interpret the resulting linear system as finite difference equations

The P1 finite element machinery results in a linear system where equation no i is

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f, \varphi_i)$$

Note:

- We have used u_i for c_i to make notation similar to finite differences
- The finite difference counterpart is just $u_i = f_i$

Rewrite the left-hand side of finite element equation no i:

$$h(u_i + \frac{1}{6}(u_{i-1} - 2u_i + u_{i+1})) = [h(u + \frac{h^2}{6}D_x D_x u)]_i$$

This is the standard finite difference approximation of

$$h(u + \frac{h^2}{6}u'')$$

$$(f, \varphi_i) = \int_{x_{i-1}}^{x_i} f(x) \frac{1}{h}(x - x_{i-1}) dx + \int_{x_i}^{x_{i+1}} f(x) \frac{1}{h}(1 - (x - x_i)) dx$$

Cannot do much unless we specialize f or use *numerical integration*.

Trapezoidal rule using the nodes:

$$(f, \varphi_i) = \int_{\Omega} f \varphi_i dx \approx h \frac{1}{2}(f(x_0)\varphi_i(x_0) + f(x_N)\varphi_i(x_N)) + h \sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j)$$

$\varphi_i(x_j) = \delta_{ij}$, so this formula collapses to one term:

$$(f, \varphi_i) \approx h f(x_i), \quad i = 1, \ldots, N - 1.$$

Same result as in collocation (interpolation) and the finite difference method!

$$\int_\Omega g(x)dx \approx \frac{h}{6}\left(g(x_0) + 2\sum_{j=1}^{N-1} g(x_j) + 4\sum_{j=0}^{N-1} g(x_{j+\frac{1}{2}}) + f(x_{2N})\right),$$

Our case: $g = f\varphi_i$. The sums collapse because $\varphi_i = 0$ at most of the points.

$$(f, \varphi_i) \approx \frac{h}{3}(f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}})$$

Conclusions:

- While the finite difference method just samples f at x_i, the finite element method applies an average (smoothing) of f around x_i
- On the left-hand side we have a term $\sim hu''$, and u'' also contribute to smoothing
- There is some inherent smoothing in the finite element method

With Trapezoidal integration of (f, φ_i), the finite element method essentially solve

$$u + \frac{h^2}{6} u'' = f, \quad u'(0) = u'(L) = 0,$$

by the finite difference method

$$[u + \frac{h^2}{6} D_x D_x u = f]_i$$

With Simpson integration of (f, φ_i) we essentially solve

$$[u + \frac{h^2}{6} D_x D_x u = \bar{f}]_i,$$

where

$$\bar{f}_i = \frac{1}{3}(f_{i-1/2} + f_i + f_{i+1/2})$$

Note: as $h \to 0$, $hu'' \to 0$ and $\bar{f}_i \to f_i$.

- Can we adjust the finite element method so that we do not get the extra hu'' smoothing term and averaging of f?
- This allows finite elements to inherit (desired) properties of finite differences

Result:

- Compute all integrals by the Trapezoidal method and P1 elements
- Specifically, the coefficient matrix becomes diagonal ("lumped") - no linear system (!)
- Loss of accuracy? The Trapezoidal rule has error $\mathcal{O}(h^2)$, the same as the approximation error in P1 elements

So far,

- *Nodes*: points for defining φ_i and computing u values
- *Elements*: subdomain (containing a few nodes)
- This is a common notion of nodes and elements

One problem:

- Our algorithms need nodes at the element boundaries
- This is often not desirable, so we need to throw the `nodes` and `elements` arrays away and find a more generalized element concept

- We introduce *cell* for the subdomain that we up to now called element
- A cell has *vertices* (interval end points)
- *Nodes* are, almost as before, points where we want to compute unknown functions
- *Degrees of freedom* is what the c_j represent (usually function values at nodes)

1. a *reference cell* in a local reference coordinate system
2. a set of *basis functions* $\tilde{\varphi}_r$ defined on the cell
3. a set of *degrees of freedom* (e.g., function values) that uniquely determine the basis functions such that $\tilde{\varphi}_r = 1$ for degree of freedom number r and $\tilde{\varphi}_r = 0$ for all other degrees of freedom
4. a mapping between local and global degree of freedom numbers (*dof map*)
5. a geometric *mapping* of the reference cell onto to cell in the physical domain: $[-1, 1] \Rightarrow [x_L, x_R]$

- Cell vertex coordinates: `vertices` (equals nodes for P1 elements)
- Element vertices: `cells[e][r]` holds global vertex number of local vertex no `r` in element e (same as `elements` for P1 elements)
- `dof_map[e,r]` maps local dof `r` in element e to global dof number (same as `elements` for Pd elements)

The assembly process now applies `dof_map`:

```
A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
b[dof_map[e][r]] += b_e[r]
```
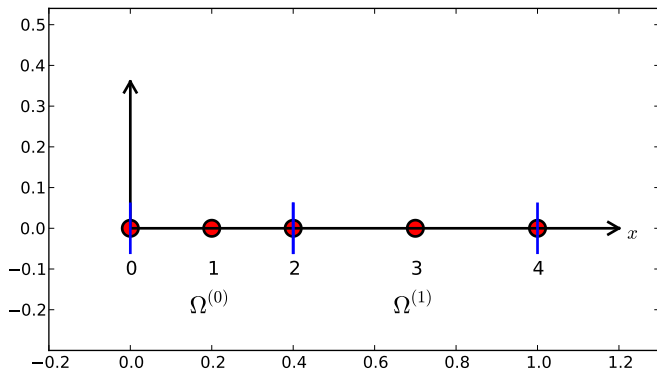
```
vertices = [0, 0.4, 1]
cells = [[0, 1], [1, 2]]
dof_map = [[0, 1, 2], [2, 3, 4]]
```

Example: Same mesh, but u is piecewise constant in each cell (P0 element). Same `vertices` and `cells`, but

```
dof_map = [[0], [1]]
```

May think of one node in the middle of each element.

Note:

We will hereafter work with `cells`, `vertices`, and `dof_map`.

```python
# Use modified fe_approx1D module
from fe_approx1D_numint import *

x = sp.Symbol('x')
f = x*(1 - x)

N_e = 10
# Create mesh with P3 (cubic) elements
vertices, cells, dof_map = mesh_uniform(N_e, d=3, Omega=[0,1])

# Create basis functions on the mesh
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]

# Create linear system and solve it
A, b = assemble(vertices, cells, dof_map, phi, f)
c = np.linalg.solve(A, b)

# Make very fine mesh and sample u(x) on this mesh for plotting
x_u, u = u_glob(c, vertices, cells, dof_map,
                resolution_per_element=51)
plot(x_u, u)
```
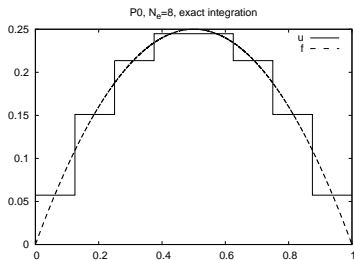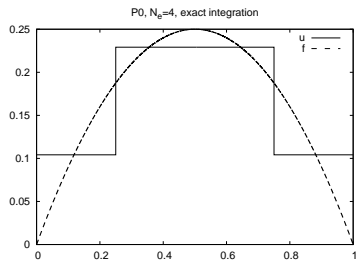
The `approximate` function automates the steps in the previous slide:

```python
from fe_approx1D_numint import *
x=sp.Symbol("x")
for N_e in 4, 8:
    approximate(x*(1-x), d=0, N_e=N_e, Omega=[0,1])
```

$$L^2 \text{ error:} \quad ||e||_{L^2} = \left(\int_\Omega e^2 \, dx \right)^{1/2}$$

Accurate approximation of the integral:

- Sample $u(x)$ at many points in each element (call `u_glob`, returns `x` and `u`)
- Use the Trapezoidal rule based on the samples
- It is important to integrate u accurately *over the elements*
- (In a finite difference method we would just sample the mesh point values)

Note

We need a version of the Trapezoidal rule valid for non-uniformly spaced points:

$$\int_\Omega g(x)dx \approx \sum_{j=0}^{n-1} \frac{1}{2}(g(x_j) + g(x_{j+1}))(x_{j+1} - x_j)$$

```
# Given c, compute x and u values on a very fine mesh
x, u = u_glob(c, vertices, cells, dof_map,
              resolution_per_element=101)
# Compute the error on the very fine mesh
e = f(x) - u
e2 = e**2
# Vectorized Trapezoidal rule
E = np.sqrt(0.5*np.sum((e2[:-1] + e2[1:])*(x[1:] - x[:-1])))
```

Theory and experiments show that the least squares or projection/Galerkin method in combination with Pd elements of equal length h has an error

$$||e||_{L^2} = Ch^{d+1}$$

where C depends on f, but not on h or d.

- Can we construct $\varphi_i(x)$ with continuous derivatives? Yes!

Consider a reference cell $[-1, 1]$. We introduce two nodes, $X = -1$ and $X = 1$. The degrees of freedom are

- 0: value of function at $X = -1$
- 1: value of first derivative at $X = -1$
- 2: value of function at $X = 1$
- 3: value of first derivative at $X = 1$

Derivatives as unknowns ensure the same $\varphi_i'(x)$ value at nodes and thereby continuous derivatives.

4 constraints on $\tilde{\varphi}_r$ (1 for dof r, 0 for all others):

- $\tilde{\varphi}_0(X_{(0)}) = 1$, $\tilde{\varphi}_0(X_{(1)}) = 0$, $\tilde{\varphi}_0'(X_{(0)}) = 0$, $\tilde{\varphi}_0'(X_{(1)}) = 0$
- $\tilde{\varphi}_1'(X_{(0)}) = 1$, $\tilde{\varphi}_1'(X_{(1)}) = 0$, $\tilde{\varphi}_1(X_{(0)}) = 0$, $\tilde{\varphi}_1(X_{(1)}) = 0$
- $\tilde{\varphi}_2(X_{(1)}) = 1$, $\tilde{\varphi}_2(X_{(0)}) = 0$, $\tilde{\varphi}_2'(X_{(0)}) = 0$, $\tilde{\varphi}_2'(X_{(1)}) = 0$
- $\tilde{\varphi}_3'(X_{(1)}) = 1$, $\tilde{\varphi}_3'(X_{(0)}) = 0$, $\tilde{\varphi}_3(X_{(0)}) = 0$, $\tilde{\varphi}_3(X_{(1)}) = 0$

This gives 4 linear, coupled equations *for each* $\tilde{\varphi}_r$ to determine the 4 coefficients in the cubic polynomial

$$\tilde{\varphi}_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \tag{19}$$

$$\tilde{\varphi}_1(X) = -(X+1)(1 - \frac{1}{2}(X+1))^2 \tag{20}$$

$$\tilde{\varphi}_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \tag{21}$$

$$\tilde{\varphi}_3(X) = -\frac{1}{2}(X+1)(\frac{1}{2}(X+1)^2 - (X+1)) \tag{22}$$

$$\tag{23}$$

- $\int_\Omega f\varphi_i dx$ must in general be computed by numerical integration
- Numerical integration is often used for the matrix too

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j g(\bar{X}_j),$$

where

- $\bar{X}_j$ are *integration points*
- w_j are *integration weights*

Different rules correspond to different choices of points and weights

Simplest possibility: the Midpoint rule,

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0, \ w_0 = 2,$$

Exact for linear integrands

- Idea: use a fixed, uniformly distributed set of points in $[-1, 1]$
- The points often coincides with nodes
- Very useful for making $\varphi_i \varphi_j = 0$ and get diagonal ("mass") matrices ("lumping")

The Trapezoidal rule:

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1, \; \bar{X}_1 = 1, \; w_0 = w_1 = 1,$$

Simpson's rule:

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right),$$

where

$$\bar{X} = 1 \quad \bar{X} = 0 \quad \bar{X} = 1 \quad w = w = \frac{1}{} \quad w = \frac{4}{}$$

- Optimize the location of points to get higher accuracy
- Gauss-Legendre rules (quadrature) adjust points and weights to integrate polynomials exactly

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}}, \ \bar{X}_1 = \frac{1}{\sqrt{3}}, \ w_0 = w_1 = 1 \tag{24}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}}, \ \bar{X}_0 = 0, \ \bar{X}_2 = \sqrt{\frac{3}{5}}, \ w_0 = w_2 = \frac{5}{9}, \ w_1 = \frac{8}{9} \tag{25}$$

- $M = 1$: integrates 3rd degree polynomials exactly
- $M = 2$: integrates 5th degree polynomials exactly
- In general, M-point rule integrates a polynomial of degree $2M + 1$ exactly.

See `numint.py` for a large collection of Gauss-Legendre rules.

Extensibility of 1D ideas.

All the concepts and algorithms developed for approximation of 1D functions $f(x)$ can readily be extended to 2D functions $f(x, y)$ and 3D functions $f(x, y, z)$. Key formulas stay the same.

Inner product in 2D:

$$(f, g) = \int_\Omega f(x, y)g(x, y)dxdy$$

Least squares and project/Galerkin lead to a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j}c_j = b_i, \quad i \in \mathcal{I}_s$$

$$A_{i,j} = (\psi_i, \psi_j)$$

$$b_i = (f, \psi_i)$$

Challenge: How to construct 2D basis functions $\psi_i(x, y)$?

Use a 1D basis for x variation and a similar for y variation:

$$V_x = \text{span}\{\hat{\psi}_0(x), \ldots, \hat{\psi}_{N_x}(x)\} \tag{26}$$

$$V_y = \text{span}\{\hat{\psi}_0(y), \ldots, \hat{\psi}_{N_y}(y)\} \tag{27}$$

The 2D vector space can be defined as a *tensor product* $V = V_x \otimes V_y$ with basis functions

$$\psi_{p,q}(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y) \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y .$$

Given two vectors $a = (a_0, \ldots, a_M)$ and $b = (b_0, \ldots, b_N)$ their *outer tensor product*, also called the *dyadic product*, is $p = a \otimes b$, defined through

$$p_{i,j} = a_i b_j, \quad i = 0, \ldots, M, \ j = 0, \ldots, N.$$

Note: p has two indices (as a matrix or two-dimensional array)

Example: 2D basis as tensor product of 1D spaces,

$$\psi_{p,q}(x, y) = \hat{\psi}_p(x) \hat{\psi}_q(y), \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y$$

The 2D basis can employ a double index and double sum:

$$u = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} c_{p,q} \psi_{p,q}(x, y)$$

Or just a single index:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x, y)$$

with an index mapping $(p, q) \rightarrow i$:

$$\psi_i(x, y) = \hat{\psi}_p(x) \hat{\psi}_q(y), \quad i = p(N_y + 1) + q \text{ or } i = q(N_x + 1) + p$$

In 1D we use the basis

$$\{1, x\}$$

2D tensor product (all combinations):

$$\psi_{0,0} = 1, \quad \psi_{1,0} = x, \quad \psi_{0,1} = y, \quad \psi_{1,1} = xy$$

or with a single index:

$$\psi_0 = 1, \quad \psi_1 = x, \quad \psi_2 = y, \quad \psi_3 = xy$$

See notes for details of a hand-calculation.

Quadratic $f(x, y) = (1 + x^2)(1 + 2y^2)$ (left), bilinear u (right):
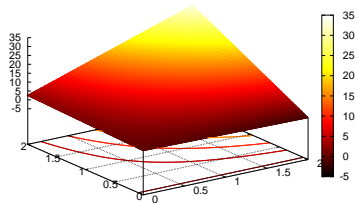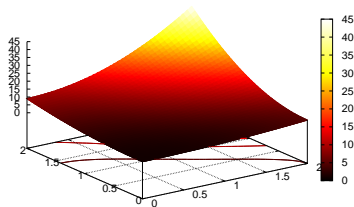
Very small modification of `approx1D.py`:

- `Omega = [[0, L_x], [0, L_y]]`
- Symbolic integration in 2D
- Construction of 2D (tensor product) basis functions

```python
import sympy as sp

integrand = psi[i]*psi[j]
I = sp.integrate(integrand,
                 (x, Omega[0][0], Omega[0][1]),
                 (y, Omega[1][0], Omega[1][1]))

# Fall back on numerical integration if symbolic integration
# was unsuccessful
if isinstance(I, sp.Integral):
    integrand = sp.lambdify([x,y], integrand)
    I = sp.mpmath.quad(integrand,
                       [Omega[0][0], Omega[0][1]],
                       [Omega[1][0], Omega[1][1]])
```

Tensor product of 1D "Taylor-style" polynomials x^i:

```python
def taylor(x, y, Nx, Ny):
    return [x**i*y**j for i in range(Nx+1) for j in range(Ny+1)]
```

Tensor product of 1D sine functions $\sin((i+1)\pi x)$:

```python
def sines(x, y, Nx, Ny):
    return [sp.sin(sp.pi*(i+1)*x)*sp.sin(sp.pi*(j+1)*y)
            for i in range(Nx+1) for j in range(Ny+1)]
```

Complete code in `approx2D.py`

$$f(x, y) = (1 + x^2)(1 + 2y^2)$$

```
>>> from approx2D import *
>>> f = (1+x**2)*(1+2*y**2)
>>> psi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
>>> u, c = least_squares(f, psi, Omega)
>>> print u
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print sp.expand(f)
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

Add higher powers to the basis such that $f \in V$:

```
>>> psi = taylor(x, y, 2, 2)
>>> u, c = least_squares(f, psi, Omega)
>>> print u
2*x**2*y**2 + x**2 + 2*y**2 + 1
>>> print u-f
0
```

Expected: $u = f$ when $f \in V$

Key idea:

$$V = V_x \otimes V_y \otimes V_z$$

Repeated outer tensor product of multiple vectors

$$a^{(q)} = (a_0^{(q)}, \ldots, a_{N_q}^{(q)}), \quad q = 0, \ldots, m$$

$$p = a^{(0)} \otimes \cdots \otimes a^{(m)}$$

$$p_{i_0, i_1, \ldots, i_m} = a_{i_1}^{(0)} a_{i_1}^{(1)} \cdots a_{i_m}^{(m)}$$

$$\psi_{p,q,r}(x, y, z) = \hat{\psi}_p(x)\hat{\psi}_q(y)\hat{\psi}_r(z)$$

$$u(x, y, z) = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} \sum_{r \in \mathcal{I}_z} c_{p,q,r} \psi_{p,q,r}(x, y, z)$$

The two great advantages of the finite element method:

- Can handle complex-shaped domains in 2D and 3D
- Can easily provide higher-order polynomials in the approximation

Finite elements in 1D: mostly for learning, insight, debugging

2D:

- triangles
- quadrilaterals

3D:

- tetrahedra
- hexahedra
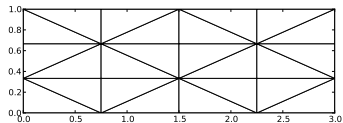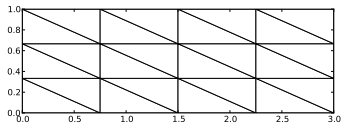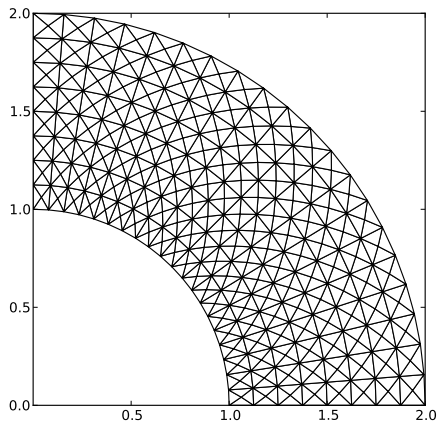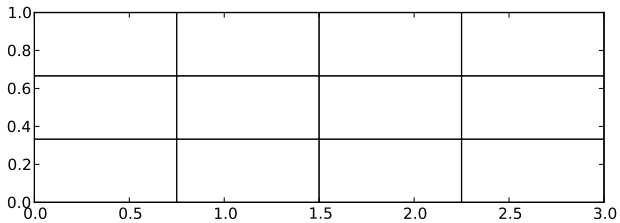
The P1 triangular 2D element: u is linear $ax + by + c$ over each triangular cell
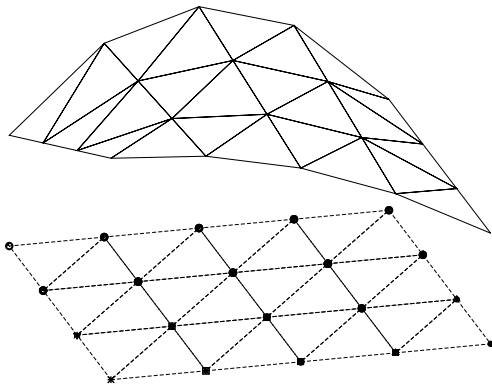
- Cells = triangles
- Vertices = corners of the cells
- Nodes = vertices
- Degrees of freedom = function values at the nodes

local   global
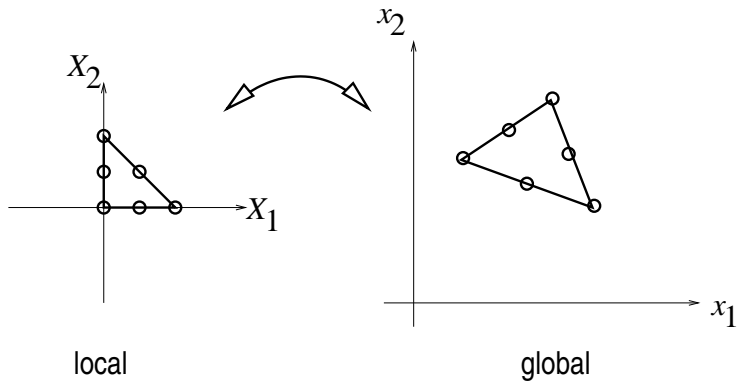
- $\varphi_i(x, y)$ varies linearly over each cell
- $\varphi_i = 1$ at vertex (node) i, 0 at all other vertices (nodes)
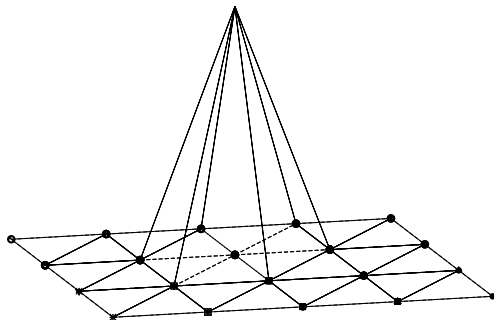
- As in 1D, the contribution from one cell to the matrix involves just a few entries, collected in the element matrix and vector
- $\varphi_i \varphi_j \neq 0$ only if i and j are degrees of freedom (vertices/nodes) in the same element
- The 2D P1 element has a 3×3 element matrix
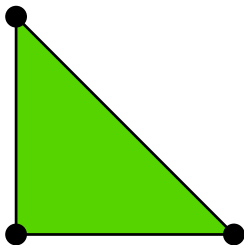
$$\tilde{\varphi}_0(X, Y) = 1 - X - Y \qquad (28)$$
$$\tilde{\varphi}_1(X, Y) = X \qquad (29)$$
$$\tilde{\varphi}_2(X, Y) = Y \qquad (30)$$

Higher-degree $\tilde{\varphi}_r$ introduce more nodes (dof = node values)
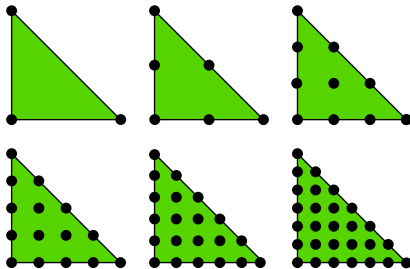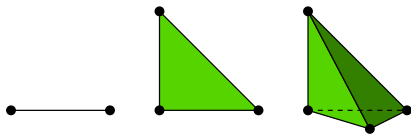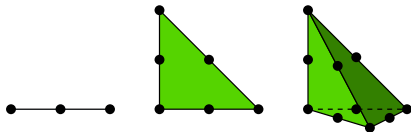
- Interval, triangle, tetrahedron: *simplex* element (plural quick-form: *simplices*)
- Side of the cell is called *face*
- Thetrahedron has also *edges*

Mapping of local $\boldsymbol{X} = (X, Y)$ coordinates in the reference cell to global, physical $\boldsymbol{x} = (x, y)$ coordinates:

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r^{(1)}(\boldsymbol{X}) \boldsymbol{x}_{q(e,r)} \tag{31}$$

where

- r runs over the local vertex numbers in the cell
- x_i are the (x, y) coordinates of vertex i
- $\tilde{\varphi}_r^{(1)}$ are P1 basis functions

This mapping preserves the straight/planar faces and edges.
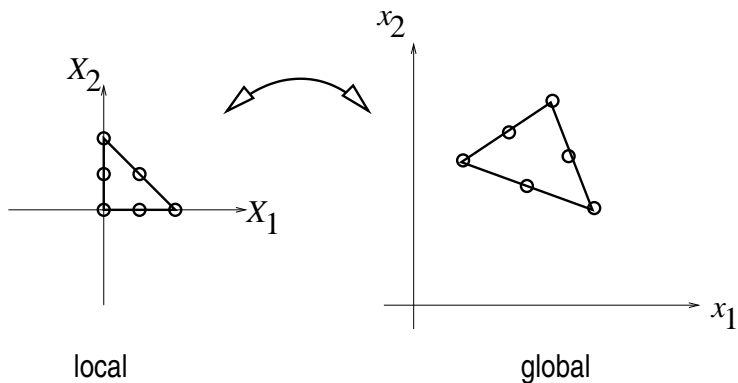
local   global

Idea: Use the basis functions of the element (not only the P1 functions) to map the element

$$x = \sum_r \tilde{\varphi}_r(X) x_{q(e,r)}$$

Advantage: higher-order polynomial basis functions now map the reference cell to a *curved* triangle or tetrahedron.

local   global
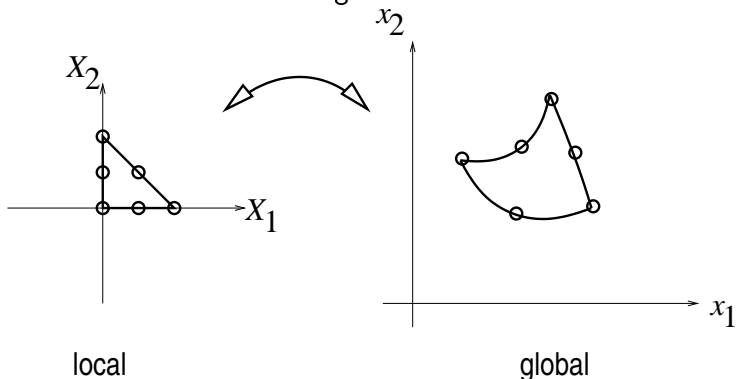
Integrals must be transformed from $\Omega^{(e)}$ (physical cell) to $\tilde{\Omega}^r$ (reference cell):

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x}) \, \mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})\tilde{\varphi}_j(\boldsymbol{X}) \det J \, \mathrm{d}X \tag{32}$$

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})f(\boldsymbol{x}) \, \mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})f(\boldsymbol{x}(\boldsymbol{X})) \det J \, \mathrm{d}X \tag{33}$$

where $\mathrm{d}x = dxdy$ or $\mathrm{d}x = dxdydz$ and $\det J$ is the determinant of the Jacobian of the mapping $\boldsymbol{x}(\boldsymbol{X})$.

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X}\frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y}\frac{\partial y}{\partial X}$$

Affine mapping (31): $\det J = 2\Delta$, $\Delta =$ cell volume

Finite elements in 2D and 3D builds on the same *ideas* and *concepts* as in 1D, but there is simply much more to compute because the specific mathematical formulas in 2D and 3D are more complicated and the book keeping with dof maps also gets more complicated. The manual work is tedious, lengthy, and error-prone so automation by the computer is a must.

Our aim is to extend the ideas for approximating f by u, or solving

$$u = f$$

to real, *spatial* differential equations like

$$-u'' + bu = f, \quad u(0) = C, \ u'(L) = D$$

Emphasis will be on the Galerkin/projection method

$$\mathcal{L}(u) = 0, \quad x \in \Omega$$

Examples (1D problems):

$$\mathcal{L}(u) = \frac{d^2 u}{dx^2} - f(x),$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) + f(x),$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) - au + f(x),$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) + f(u, x)$$

$$\mathcal{B}_0(u) = 0, \ x = 0, \quad \mathcal{B}_1(u) = 0, \ x = L$$

Examples:

$$\mathcal{B}_i(u) = u - g, \qquad\qquad\qquad \text{Dirichlet condition}$$

$$\mathcal{B}_i(u) = -\alpha\frac{du}{dx} - g, \qquad\qquad \text{Neumann condition}$$

$$\mathcal{B}_i(u) = -\alpha\frac{du}{dx} - h(u - g), \qquad \text{Robin condition}$$

- $u_e(x)$ is the symbol for the *exact* solution of $\mathcal{L}(u_e) = 0 +$ $\mathcal{B}_i = 0$
- $u(x)$ denotes an *approximate* solution
- $V = \text{span}\{\psi_0(x), \ldots, \psi_N(x)\}$, V has basis $\{\psi_i\}_{i \in \mathcal{I}_s}$
- We seek $u \in V$
- $\mathcal{I}_s = \{0, \ldots, N\}$ is an index set
- $u(x) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$
- Inner product: $(u, v) = \int_\Omega uv \, \mathrm{d}x$
- Norm: $||u|| = \sqrt{(u, u)}$

Much is similar to approximating a function (solving $u = f$), but two new topics are needed:

- Variational formulation of the differential equation problem (including integration by parts)
- Handling of boundary conditions

- When solving $u = f$ we knew the error $e = f - u$ and could use principles for minimizing the error
- When solving $\mathcal{L}(u_e) = 0$ we do not know u_e and cannot work with the error $e = u_e - u$
- We can only know the *error in the equation*: the residual R

Inserting $u = \sum_j c_j \psi_j$ in $\mathcal{L} = 0$ gives a residual R

$$\mathcal{L}(u) = \mathcal{L}(\sum_j c_j \psi_j) = R \neq 0$$

Goal: minimize R with respect to $\{c_i\}_{i \in \mathcal{I}_s}$ (and hope it makes a small e too)

$$R = R(c_0, \ldots, c_N; x)$$

Idea: minimize

$$E = ||R||^2 = (R, R) = \int_\Omega R^2 dx$$

Minimization wrt $\{c_i\}_{i \in \mathcal{I}_s}$ implies

$$\frac{\partial E}{\partial c_i} = \int_\Omega 2R \frac{\partial R}{\partial c_i} dx = 0 \quad \Leftrightarrow \quad (R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s$$

$N + 1$ equations for $N + 1$ unknowns $\{c_i\}_{i \in \mathcal{I}_s}$

Idea: make R orthogonal to V,

$$(R, v) = 0, \quad \forall v \in V$$

This implies

$$(R, \psi_i) = 0, \quad i \in \mathcal{I}_s$$

$N + 1$ equations for $N + 1$ unknowns $\{c_i\}_{i \in \mathcal{I}_s}$

Generalization of the Galerkin method: demand R orthogonal to some space W, possibly $W \neq V$:

$$(R, v) = 0, \quad \forall v \in W$$

If $\{w_0, \ldots, w_N\}$ is a basis for W:

$$(R, w_i) = 0, \quad i \in \mathcal{I}_s$$

- $N + 1$ equations for $N + 1$ unknowns $\{c_i\}_{i \in \mathcal{I}_s}$
- Weighted residual with $w_i = \partial R / \partial c_i$ gives least squares

- ψ_j used in $\sum_j c_j \psi_j$ is called *trial function*
- ψ_i or w_i used as weight in Galerkin's method is called *test function*

The collocation method

Idea: demand $R = 0$ at $N + 1$ points in space

$$R(x_i; c_0, \ldots, c_N) = 0, \quad i \in \mathcal{I}_s$$

The collocation method is a weighted residual method with delta functions as weights

$$0 = \int_\Omega R(x; c_0, \ldots, c_N)\delta(x - x_i)\, \mathrm{d}x = R(x_i; c_0, \ldots, c_N)$$

property of $\delta(x)$: $\quad \int_\Omega f(x)\delta(x - x_i)dx = f(x_i), \quad x_i \in \Omega$
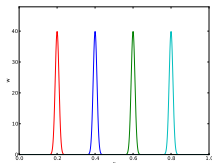
Goal

Exemplify the least squares, Galerkin, and collocation methods in a simple 1D problem with global basis functions.

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0, \ u(L) = 0$$

Basis functions:

$$\psi_i(x) = \sin\left((i+1)\pi\frac{x}{L}\right), \quad i \in \mathcal{I}_s$$

Residual:

$$
\begin{aligned}
R(x; c_0, \ldots, c_N) &= u''(x) + f(x), \\
&= \frac{d^2}{dx^2}\left(\sum_{j \in \mathcal{I}_s} c_j \psi_j(x)\right) + f(x), \\
&= -\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x)
\end{aligned}
$$

Since $u(0) = u(L) = 0$ we must ensure that all $\psi_i(0) = \psi_i(L) = 0$, because then

$$u(0) = \sum_j c_j \psi_j(0) = 0, \quad u(L) = \sum_j c_j \psi_j(L) = 0$$

- u known: Dirichlet boundary condition
- u' known: Neumann boundary condition
- Must have $\psi_i = 0$ where Dirichlet conditions apply

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s$$

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i} \left(\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x) \right) = \psi_i''(x)$$

Because:

$$\frac{\partial}{\partial c_i} \left(c_0 \psi_0'' + c_1 \psi_1'' + \cdots + c_{i-1} \psi_{i-1}'' + c_i \psi_i'' + c_{i+1} \psi_{i+1}'' + \cdots + c_N \psi_N'' \right) =$$

$$\left(\sum_j c_j \psi_j'' + f, \psi_i''\right) = 0, \quad i \in \mathcal{I}_s$$

Rearrangement:

$$\sum_{j \in \mathcal{I}_s} (\psi_i'', \psi_j'') c_j = -(f, \psi_i''), \quad i \in \mathcal{I}_s$$

This is a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s$$

$$A_{i,j} = (\psi_i'', \psi_j'')$$

$$= \pi^4 (i+1)^2 (j+1)^2 L^{-4} \int_0^L \sin\left((i+1)\pi\frac{x}{L}\right) \sin\left((j+1)\pi\frac{x}{L}\right) \, dx$$

$$= \left\{ \begin{array}{ll} \frac{1}{2} L^{-3} \pi^4 (i+1)^4 & i = j \\ 0, & i \neq j \end{array} \right.$$

$$b_i = -(f, \psi_i'') = (i+1)^2 \pi^2 L^{-2} \int_0^L f(x) \sin\left((i+1)\pi\frac{x}{L}\right) \, dx$$

Useful property of the chosen basis functions:

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right) \sin\left((j+1)\pi\frac{x}{L}\right) dx = \delta_{ij}, \qquad \delta_{ij} = \left\{ \begin{array}{ll} \frac{1}{2}L & i = j \\ 0, & i \neq j \end{array} \right.$$

$\Rightarrow (\psi_i'', \psi_j'') = \delta_{ij}$, i.e., diagonal $A_{i,j}$, and we can easily solve for c_i:

$$c_i = \frac{2L}{\pi^2(i+1)^2} \int_0^L f(x) \sin\left((i+1)\pi\frac{x}{L}\right) dx$$

Let sympy do the work ($f(x) = 2$):

```python
from sympy import *
import sys

i, j = symbols('i j', integer=True)
x, L = symbols('x L')
f = 2
a = 2*L/(pi**2*(i+1)**2)
c_i = a*integrate(f*sin((i+1)*pi*x/L), (x, 0, L))
c_i = simplify(c_i)
print c_i
```

$$c_i = 4\frac{L^2\left((-1)^i + 1\right)}{\pi^3\left(i^3 + 3i^2 + 3i + 1\right)}, \quad u(x) = \sum_{k=0}^{N/2}\frac{8L^2}{\pi^3(2k+1)^3}\sin\left((2k+1)\pi\frac{x}{L}\right)$$

Fast decay: $c_2 = c_0/27$, $c_4 = c_0/125$ - only one term might be good enough:

$$u(x) \approx \frac{8L^2}{\pi^3}\sin\left(\pi\frac{x}{L}\right)$$

$R = u'' + f$:

$$(u'' + f, v) = 0, \quad \forall v \in V,$$

or rearranged,

$$(u'', v) = -(f, v), \quad \forall v \in V$$

This is a *variational formulation* of the differential equation problem.

$\forall v \in V$ is equivalent with $\forall v \in \psi_i$, $i \in \mathcal{I}_s$, resulting in

$$\left(\sum_{j \in \mathcal{I}_s} c_j \psi_j'', \psi_i \right) = -(f, \psi_i), \quad i \in \mathcal{I}_s$$

$$\sum_{j \in \mathcal{I}_s} (\psi_j'', \psi_i) c_j = -(f, \psi_i), \quad i \in \mathcal{I}_s$$

Since $\psi_i'' \propto -\psi_i$, Galerkin's method gives the same linear system and the same solution as the least squares method (in this particular example).

$R = 0$ (i.e.,the differential equation) must be satisfied at $N + 1$ points:

$$-\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x_i) = f(x_i), \quad i \in \mathcal{I}_s$$

This is a linear system $\sum_j A_{i,j} = b_i$ with entries

$$A_{i,j} = -\psi_j''(x_i) = (j+1)^2 \pi^2 L^{-2} \sin\left((j+1)\pi\frac{x_i}{L}\right), \quad b_i = 2$$

Choose: $N = 0$, $x_0 = L/2$

$$c_0 = 2L^2/\pi^2$$

- Exact solution: $u(x) = x(L - x)$
- Galerkin or least squares ($N = 0$): $u(x) = 8L^2\pi^{-3}\sin(\pi x/L)$
- Collocation method ($N = 0$): $u(x) = 2L^2\pi^{-2}\sin(\pi x/L)$.

```
>>> import sympy as sp
>>> # Computing with Dirichlet conditions: -u''=2 and sines
>>> x, L = sp.symbols('x L')
>>> e_Galerkin = x*(L-x) - 8*L**2*sp.pi**(-3)*sp.sin(sp.pi*x/L)
>>> e_colloc = x*(L-x) - 2*L**2*sp.pi**(-2)*sp.sin(sp.pi*x/L)

>>> # Verify max error for x=L/2
>>> dedx_Galerkin = sp.diff(e_Galerkin, x)
>>> dedx_Galerkin.subs(x, L/2)
0
>>> dedx_colloc = sp.diff(e_colloc, x)
>>> dedx_colloc.subs(x, L/2)
0

# Compute max error: x=L/2, evaluate numerical, and simplify
>>> sp.simplify(e_Galerkin.subs(x, L/2).evalf(n=3))
-0.00812*L**2
>>> sp.simplify(e_colloc.subs(x, L/2).evalf(n=3))
0.0473*L**2
```

Second-order derivatives will hereafter be integrated by parts

$$\int_0^L u''(x)v(x)dx = -\int_0^L u'(x)v'(x)dx + [vu']_0^L$$

$$= -\int_0^L u'(x)v'(x)dx + u'(L)v(L) - u'(0)v(0)$$

Motivation:

- Lowers the order of derivatives
- Gives more symmetric forms (incl. matrices)
- Enables easy handling of Neumann boundary conditions
- Finite element basis functions φ_i have discontinuous derivatives (at cell boundaries) and are not suited for terms with φ_i''

We use a boundary function to deal with non-zero Dirichlet boundary conditions

- What about nonzero Dirichlet conditions? Say $u(L) = D$
- We always require $\psi_i(L) = 0$ (i.e., $\psi_i = 0$ where Dirichlet conditions applies)
- Problem: $u(L) = \sum_j c_j \psi_j(L) = \sum_j c_j \cdot 0 = 0 \neq D$ - always!
- Solution: $u(x) = B(x) + \sum_j c_j \psi_j(x)$
- $B(x)$: user-constructed boundary function that fulfills the Dirichlet conditions
- If $u(L) = D$, make sure $B(L) = D$
- No restrictions of how $B(x)$ varies in the interior of Ω

Dirichlet conditions: $u(0) = C$ and $u(L) = D$. Choose for example

$$B(x) = \frac{1}{L}(C(L - x) + Dx) : \qquad B(0) = C, \ B(L) = D$$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x),$$

$$u(0) = B(0) = C, \quad u(L) = B(L) = D$$

Dirichlet condition: $u(L) = D$. Choose for example

$$B(x) = D : \qquad B(L) = D$$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x),$$

$$u(L) = B(L) = D$$

- $\{\psi_i\}_{i \in \mathcal{I}_s}$ is a basis for V
- $\sum_{j \in \mathcal{I}_s} c_j \psi_j(x) \in V$
- But $u \notin V$!
- Reason: say $u(0) = C$ and $u \in V$ (any $v \in V$ has $v(0) = C$, then $2u \notin V$ because $2u(0) = 2C$
- When $u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$, $B \neq 0$, $B \notin V$ (in general) and $u \notin V$, but $(u - B) \in V$ since $\sum_j c_j \psi_j \in V$

The finite element literature (and much FEniCS documentation) applies an abstract notation for the variational formulation:

Find $(u - B) \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in V$$

$$-u'' = f, \quad u'(0) = C, \ u(L) = D, \quad u = D + \sum_j c_j \psi_j$$

Variational formulation:

$$\int_\Omega u'v'dx = \int_\Omega fvdx - v(0)C \quad \text{or} \quad (u', v') = (f, v) - v(0)C \quad \forall v \in V$$

Abstract formulation: find $(u - B) \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in V$$

We identify

$$a(u, v) = (u', v'), \quad L(v) = (f, v) - v(0)C$$

- $a(u, v)$ is a *bilinear form*
- $L(v)$ is a *linear form*

Linear form means

$$L(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 L(v_1) + \alpha_2 L(v_2),$$

Bilinear form means

$$a(\alpha_1 u_1 + \alpha_2 u_2, v) = \alpha_1 a(u_1, v) + \alpha_2 a(u_2, v),$$
$$a(u, \alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 a(u, v_1) + \alpha_2 a(u, v_2)$$

In nonlinear problems: Find $(u - B) \in V$ such that
$F(u; v) = 0 \ \forall v \in V$

$$a(u, v) = L(v) \quad \forall v \in V \quad \Leftrightarrow \quad a(u, \psi_i) = L(\psi_i) \quad i \in \mathcal{I}_s$$

We can now derive the corresponding linear system once and for all by inserting $u = B + \sum_j c_j \psi_j$:

$$a(B + \sum_{j \in \mathcal{I}_s} c_j \psi_j, \psi_i) c_j = L(\psi_i) \quad i \in \mathcal{I}_s$$

Because of linearity,

$$\sum_{j \in \mathcal{I}_s} \underbrace{a(\psi_j, \psi_i)}_{A_{i,j}} c_j = \underbrace{L(\psi_i) - a(B, \psi_i)}_{b_i} \quad i \in \mathcal{I}_s$$

If a is symmetric: $a(u, v) = a(v, u)$,

$$a(u, v) = L(v) \quad \forall v \in V$$

is equivalent to minimizing the functional

$$F(v) = \frac{1}{2} a(v, v) - L(v)$$

over all functions $v \in V$. That is,

$$F(u) \leq F(v) \quad \forall v \in V$$

- Much used in the early days of finite elements
- Still much used in structural analysis and elasticity
- Not as general as Galerkin's method (since we require $a(u, v) = a(v, u)$)

Goal

Derive variational formulations for some prototype differential equations in 1D that include

- variable coefficients
- mixed Dirichlet and Neumann conditions
- nonlinear coefficients

$$-\frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) = f(x), \quad x \in \Omega = [0, L], \ u(0) = C, \ u(L) = D$$

- Variable coefficient $\alpha(x)$
- $V = \text{span}\{\psi_0, \ldots, \psi_N\}$
- *Nonzero* Dirichlet conditions at $x = 0$ *and* $x = L$
- Must have $\psi_i(0) = \psi_i(L) = 0$
- Any $v \in V$ has then $v(0) = v(L) = 0$
- $B(x) = C + \frac{1}{L}(D - C)x$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_i(x),$$

$$R = -\frac{d}{dx}\left(a\frac{du}{dx}\right) - f$$

Galerkin's method:

$$(R, v) = 0, \quad \forall v \in V$$

or with integrals:

$$\int_\Omega \left(\frac{d}{dx}\left(\alpha\frac{du}{dx}\right) - f\right) v \, \mathrm{d}x = 0, \quad \forall v \in V$$

$$-\int_\Omega \frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right)v\,\mathrm{d}x = \int_\Omega \alpha(x)\frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x - \left[\alpha\frac{du}{dx}v\right]_0^L$$

Boundary terms vanish since $v(0) = v(L) = 0$

Variational formulation

Find $(u - B) \in V$ such that

$$\int_{\Omega} \alpha(x) \frac{du}{dx} \frac{dv}{dx} dx = \int_{\Omega} f(x) v dx, \quad \forall v \in V$$

Compact notation:

$$\underbrace{(\alpha u', v')}_{a(u,v)} = \underbrace{(f, v)}_{L(v)}, \quad \forall v \in V$$

With

$$a(u, v) = (\alpha u', v'), \quad L(v) = (f, v)$$

we can just use the formula for the linear system:

$$A_{i,j} = a(\psi_j, \psi_i) = (\alpha \psi_j', \psi_i') = \int_\Omega \alpha \psi_j' \psi_i' \, \mathrm{d}x = \int_\Omega \psi_i' \alpha \psi_j' \, \mathrm{d}x \quad (= a(\psi_i, \psi_j))$$

$$b_i = (f, \psi_i) - (\alpha B', \psi_i) = \int_\Omega (f \psi_i - \alpha L^{-1}(D - C)\psi_i') \, \mathrm{d}x$$

$v = \psi_i$ and $u = B + \sum_j c_j \psi_j$:

$$\left(\alpha B' + \alpha \sum_{j \in \mathcal{I}_s} c_j \psi_j', \psi_i'\right) = (f, \psi_i), \quad i \in \mathcal{I}_s$$

Reorder to form linear system:

$$\sum_{j \in \mathcal{I}_s} (\alpha \psi_j', \psi_i') c_j = (f, \psi_i) + (aL^{-1}(D - C), \psi_i'), \quad i \in \mathcal{I}_s$$

This is $\sum_j A_{i,j} c_j = b_i$ with

$$A_{i,j} = (a\psi_j', \psi_i') = \int_\Omega \alpha(x)\psi_j'(x)\psi_i'(x)\,\mathrm{d}x$$

$$b_i = (f, \psi_i) + (aL^{-1}(D - C), \psi_i') = \int_\Omega \left(f\psi_i + \alpha\frac{D - C}{L}\psi_i'\right)\,\mathrm{d}x$$

$$-u''(x) + bu'(x) = f(x), \quad x \in \Omega = [0, L], \; u(0) = C, \; u'(L) = E$$

New features:

- first-order derivative u' in the equation
- boundary condition with u': $u'(L) = E$

Initial steps:

- Must force $\psi_i(0) = 0$ because of Dirichlet condition at $x = 0$
- Boundary function: $B(x) = C(L - x)$ or just $B(x) = C$
- No requirements on $\psi_i(L)$ (no Dirichlet condition at $x = L$)

First-order derivative in the equation and boundary condition; details

$$u = C + \sum_{j \in \mathcal{I}_s} c_j \psi_i(x)$$

Galerkin's method: multiply by v, integrate over Ω, integrate by parts.

$$(-u'' + bu' - f, v) = 0, \quad \forall v \in V$$

$$(u', v') + (bu', v) = (f, v) + [u'v]_0^L, \quad \forall v \in V$$

$[u'v]_0^L = u'(L)v(L) - u'(0)v(0) = Ev(L)$ since $v(0) = 0$ and $u'(L) = E$

$$(u'v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V$$

$$(u'v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V$$

Important observations:

- The boundary term can be used to implement Neumann conditions
- Forgetting the boundary term implies the condition $u' = 0$ (!)
- Such conditions are called *natural boundary conditions*

Abstract notation:

$$a(u, v) = L(v) \quad \forall v \in V$$

Here:

$$a(u, v) = (u', v') + (bu', v)$$
$$L(v) = (f, v) + Ev(L)$$

First-order derivative in the equation and boundary condition; linear system

Insert $u = C + \sum_j c_j \psi_j$ and $v = \psi_i$ in

$$(u'v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V$$

and manipulate to get

$$\sum_{j \in \mathcal{I}_s} \underbrace{((\psi_j', \psi_i') + (b\psi_j', \psi_i))}_{A_{i,j}} c_j = \underbrace{(f, \psi_i) + E\psi_i(L)}_{b_i}, \quad i \in \mathcal{I}_s$$

Observation: $A_{i,j}$ is not symmetric because of the term

$$(b\psi_j', \psi_i) = \int_\Omega b\psi_j'\psi_i dx \neq \int_\Omega b\psi_i'\psi_j dx = (\psi_i', b\psi_j)$$

$$(u', v') + (bu', v) = (f, v) + u'(L)v(L) - u'(0)v(0)$$

- Note: forgetting the boundary terms implies $u'(L) = u'(0) = 0$ (unless prescribe a Dirichlet condition)
- Conditions on u' are simply inserted in the variational form and called *natural conditions*
- Conditions on u at $x = 0$ requires modifying V (through $\psi_i(0) = 0$) and are known as *essential conditions*

Lesson learned

It is easy to forget the boundary term when integrating by parts. That mistake may prescribe a condition on u'!

Problem:

$$-(\alpha(u)u')' = f(u), \quad x \in [0, L], \ u(0) = 0, \ u'(L) = E$$

- V: basis $\{\psi_i\}_{i \in \mathcal{I}_s}$ with $\psi_i(0) = 0$ because of $u(0) = 0$
- How does the nonlinear coefficients $\alpha(u)$ and $f(u)$ impact the variational formulation?
- (Not much!)

Galerkin: multiply by v, integrate, integrate by parts

$$\int_0^L \alpha(u) \frac{du}{dx} \frac{dv}{dx} \, dx = \int_0^L f(u) v \, dx + [\alpha(u) v u']_0^L \quad \forall v \in V$$

- $\alpha(u(0)) v(0) u'(0) = 0$ since $v(0)$
- $\alpha(u(L)) v(L) u'(L) = \alpha(u(L)) v(L) E$ since $u'(L) = E$

$$\int_0^L \alpha(u) \frac{du}{dx} \frac{dv}{dx} v \, dx = \int_0^L f(u) v \, dx + \alpha(u(L)) v(L) E \quad \forall v \in V$$

or

$$(\alpha(u) u', v') = (f(u), v) + \alpha(u(L)) v(L) E \quad \forall v \in V$$

- Abstract notation: no $a(u, v)$ and $L(v)$ because a and L get nonlinear
- Abstract notation for nonlinear problems: $F(u; v) = 0 \ \forall v \in V$
- What about forming a linear system? We get a *nonlinear* system of algebraic equations
- Must use methods like Picard iteration or Newton's method to solve nonlinear algebraic equations
- But: the variational formulation was not much affected by nonlinearities

$$-u''(x) = f(x), \quad x \in \Omega = [0, 1], \quad u'(0) = C, \; u(1) = D$$

- Use a *global* polynomial basis $\psi_i \sim x^i$ on $[0, 1]$
- Because of $u(1) = D$: $\psi_i(1) = 0$
- Basis: $\psi_i(x) = (1 - x)^{i+1}, \quad i \in \mathcal{I}_s$
- Boundary function: $B(x) = Dx$
- $u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_j = Dx + \sum_{j \in \mathcal{I}_s} c_j (1 - x)^{i+1}$

Variational formulation: find $(u - B) \in V$ such that

$$(u, \psi_i') = (f, \psi_i) - (B', \psi_i) - C\psi_i(0), \; i \in \mathcal{I}_s$$

Insert $u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_j$ and derive

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s$$

with

$$A_{i,j} = (\psi_j', \psi_i')$$
$$b_i = (f, \psi_i) - (D, \psi_i') - C\psi_i(0)$$

$$A_{i,j} = (\psi'_j, \psi'_i) = \int_0^1 \psi'_i(x)\psi'_j(x)dx = \int_0^1 (i+1)(j+1)(1-x)^{i+j} dx$$

Choose $f(x) = 2$:

$$b_i = (2, \psi_i) - (D, \psi'_i) - C\psi_i(0)$$
$$= \int_0^1 \left(2(1-x)^{i+1} - D(i+1)(1-x)^i\right) dx - C\psi_i(0)$$

Can easily do the integrals with sympy. $N = 1$ and $\mathcal{I}_s = \{0, 1\}$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 4/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} -C + D + 1 \\ 2/3 - C + D \end{pmatrix}$$

$$c_0 = -C + D + 2, \quad c_1 = -1,$$

$$u(x) = 1 - x^2 + D + C(x - 1) \quad \text{(exact solution)}$$

Assume that apart from boundary conditions, u_e lies in the same space V as where we seek u:

$$u = B + F, \quad F \in V$$
$$a(B + F, v) = L(v), \quad \forall v \in V$$
$$u_e = B + E, \quad E \in V$$
$$a(B + E, v) = L(v), \quad \forall v \in V$$

Subtract: $a(F - E, v) = 0 \implies E = F$ and $u = u_e$

Tasks:

- Address the model problem $-u''(x) = 2$, $u(0) = u(L) = 0$
- Uniform finite element mesh with P1 elements
- Show all finite element computations in detail

$$-u''(x) = 2, \quad x \in (0, L), \ u(0) = u(L) = 0,$$

Variational formulation:

$$(u', v') = (2, v) \quad \forall v \in V$$

How to deal with the boundary conditions?

Since $u(0) = 0$ and $u(L) = 0$, we must force

$$v(0) = v(L) = 0, \quad \psi_i(0) = \psi_i(L) = 0$$

Now we choose the finite element basis: $\psi_i = \varphi_i,\ i = 0, \ldots, N_n$

Problem: $\varphi_0(0) \neq 0$ and $\varphi_{N_n}(L) \neq 0$

Solution: we just exclude φ_0 and φ_{N_n} from the basis and work with

$$\psi_i = \varphi_{i+1}, \quad i = 0, \ldots, N = N_n - 2$$

Introduce index mapping $\nu(i)$: $\psi_i = \varphi_{\nu(i)}$

$$u = \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}, \quad i = 0, \ldots, N, \quad \nu(j) = j + 1$$

Irregular numbering: more complicated $\nu(j)$ table

$$A_{i,j} = \int_0^L \varphi'_{i+1}(x)\varphi'_{j+1}(x)dx, \quad b_i = \int_0^L 2\varphi_{i+1}(x)dx$$

Many will prefer to change indices to obtain a $\varphi'_i\varphi'_j$ product:
$i + 1 \rightarrow i,\ j + 1 \rightarrow j$

$$A_{i-1,j-1} = \int_0^L \varphi'_i(x)\varphi'_j(x)\,\mathrm{d}x, \quad b_{i-1} = \int_0^L 2\varphi_i(x)\,\mathrm{d}x$$
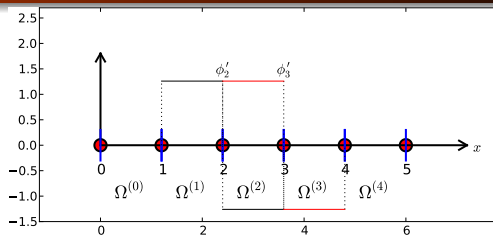
$$\varphi'_i \sim \pm h^{-1}$$

$$A_{i-1,i-1} = h^{-2}2h = 2h^{-1}, \quad A_{i-1,i-2} = h^{-1}(-h^{-1})h = -h^{-1}$$

and $A_{i-1,i} = A_{i-1,i-2}$

$$b_{i-1} = 2(\frac{1}{2}h + \frac{1}{2}h) = 2h$$

$$\frac{1}{h}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \end{pmatrix}$$

General equation at node i:

$$-\frac{1}{h}c_{i-1} + \frac{2}{h}c_i - \frac{1}{h}c_{i+1} = 2h$$

Now, $c_i = u(x_{i+1}) \equiv u_{i+1}$. Writing out the equation at node $i-1$,

$$-\frac{1}{h}c_{i-2} + \frac{2}{h}c_{i-1} - \frac{1}{h}c_i = 2h$$

translates directly to

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h$$

The standard finite difference method for $-u'' = 2$ is

$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = 2$$

Multiply by h!

The finite element method and the finite difference method are identical *in this example*.

(Remains to study the equations at the end points, which involve boundary values - but these are also the same for the two methods)

- Repeat the previous example, but apply the cellwise algorithm
- Work with one cell at a time
- Transform physical cell to reference cell $X \in [-1, 1]$

$$A_{i-1,j-1}^{(e)} = \int_{\Omega^{(e)}} \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{d}{dx}\tilde{\varphi}_r(X)\frac{d}{dx}\tilde{\varphi}_s(X)\frac{h}{2}\,\mathrm{d}X,$$

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X), \quad \tilde{\varphi}_1(X) = \frac{1}{2}(1 + X)$$

$$\frac{d\tilde{\varphi}_0}{dX} = -\frac{1}{2}, \quad \frac{d\tilde{\varphi}_1}{dX} = \frac{1}{2}$$

From the chain rule

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX}\frac{dX}{dx} = \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}$$

$$A_{i-1,j-1}^{(e)} = \int_{\Omega^{(e)}} \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_s}{dX}\frac{h}{2}\,\mathrm{d}X = \tilde{A}_{r,s}^{(e)}$$

$$b_{i-1}^{(e)} = \int_{\Omega^{(e)}} 2\varphi_i(x)\,\mathrm{d}x = \int_{-1}^{1} 2\tilde{\varphi}_r(X)\frac{h}{2}\,\mathrm{d}X = \tilde{b}_r^{(e)}, \quad i = q(e,r),\ r = 0,1$$

Must run through all $r, s = 0, 1$ and $r = 0, 1$ and compute each entry in the element matrix and vector:

$$\tilde{A}^{(e)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Example:

$$\tilde{A}_{0,1}^{(e)} = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_0}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_1}{dX}\frac{h}{2}\,\mathrm{d}X = \frac{2}{h}(-\frac{1}{2})\frac{2}{h}\frac{1}{2}\frac{h}{2}\int_{-1}^{1}\,\mathrm{d}X = -\frac{1}{h}$$

- The boundary cells involve only one unknown
- $\Omega^{(0)}$: left node value known, only a contribution from right node
- $\Omega^{(N_e)}$: right node value known, only a contribution from left node

For $e = 0$ and $= N_e$:

$$\tilde{A}^{(e)} = \frac{1}{h} \left(\ 1 \ \right), \quad \tilde{b}^{(e)} = h \left(\ 1 \ \right)$$

Only one degree of freedom ("node") in these cells ($r = 0$ counts the only dof)

4 P1 elements:

```
vertices = [0, 0.5, 1, 1.5, 2]
cells = [[0, 1], [1, 2], [2, 3], [3, 4]]
dof_map = [[0], [0, 1], [1, 2], [2]]          # only 1 dof in elm 0, 3
```

Python code for the assembly algorithm:

```
# Ae[e][r,s]: element matrix, be[e][r]: element vector
# A[i,j]: coefficient matrix, b[i]: right-hand side

for e in range(len(Ae)):
    for r in range(Ae[e].shape[0]):
        for s in range(Ae[e].shape[1]):
            A[dof_map[e,r],dof_map[e,s]] += Ae[e][i,j]
        b[dof_map[e,r]] += be[e][i,j]
```

Result: same linear system as arose from computations in the
physical domain

- Now we address nonzero Dirichlet conditions
- $B(x)$ is not always easy to construct (i.e., extend to the interior of Ω), especially not in 2D and 3D
- With finite element φ_i, $B(x)$ can be constructed in a completely general way
- I_b: set of indices with nodes where u is known
- U_i: Dirichlet value of u at node i, $i \in I_b$

The general formula for B is now

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x)$$

Suppose we have a Dirichlet condition $u(x_k) = U_k,\ k \in I_b$:

$$u(x_k) = \sum_{j \in I_b} U_j \underbrace{\varphi_j(x)}_{\neq 0 \text{ only for } j=k} + \sum_{j \in \mathcal{I}_s} c_j \underbrace{\varphi_{\nu(j)}(x_k)}_{=0,\ k \notin \mathcal{I}_s} = U_k$$

$$-u'' = 2, \quad u(0) = C, \; u(L) = D$$

$$\int_0^L u'v' \, \mathrm{d}x = \int_0^L 2v \, \mathrm{d}x \quad \forall v \in V$$

$$(u', v') = (2, v) \quad \forall v \in V$$

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x)$$

Here $I_b = \{0, N_n\}$, $U_0 = C$, $U_{N_n} = D$; ψ_i are the internal φ_i functions:

$$\psi_i = \varphi_{\nu(i)}, \quad \nu(i) = i + 1, \quad i \in \mathcal{I}_s = \{0, \ldots, N = N_n - 2\}$$

$$u(x) = \underbrace{C \cdot \varphi_0 + D\varphi_{N_n}}_{B(x)} + \sum_{j \in \mathcal{I}_s} c_j \varphi_{j+1}$$

$$= C \cdot \varphi_0 + D\varphi_{N_n} + c_0\varphi_1 + c_1\varphi_2 + \cdots + c_N\varphi_{N_n - 1}$$

Insert $u = B + \sum_j c_j \psi_j$ in variational formulation:

$$(u', v') = (2, v) \quad \Rightarrow \quad (\sum_j c_j \psi_j', \psi_i') = (2 - B', \psi_i) \quad \forall v \in V$$

$$A_{i-1,j-1} = \int_0^L \varphi_i'(x) \varphi_j'(x) \, \mathrm{d}x$$

$$b_{i-1} = \int_0^L (f(x) - C\varphi_0'(x) - D\varphi_{N_n}'(x)) \varphi_i(x) \, \mathrm{d}x$$

for $i, j = 1, \ldots, N + 1 = N_n - 1$.

New boundary terms from $-\int B' \varphi_i \, \mathrm{d}x$: $C/2$ for $i = 1$ and $-D/2$ for $i = N_n - 1$

- All element matrices are as in the previous example
- New element vector in the first and last cell

From the last cell:

$$\tilde{b}_0^{(N_e)} = \int_{-1}^{1} \left(f - D\frac{2}{h}\frac{d\tilde{\varphi}_1}{dX} \right) \tilde{\varphi}_0 \frac{h}{2} \, \mathrm{d}X = (\frac{h}{2}(2 - D\frac{2}{h}\frac{1}{2}) \int_{-1}^{1} \tilde{\varphi}_0 \, \mathrm{d}X = h - \frac{D}{2}$$

From the first cell:

$$\tilde{b}_0^{(0)} = \int_{-1}^{1} \left(f - C\frac{2}{h}\frac{d\tilde{\varphi}_0}{dX} \right) \tilde{\varphi}_1 \frac{h}{2} \, \mathrm{d}X = (\frac{h}{2}(2 + C\frac{2}{h}\frac{1}{2}) \int_{-1}^{1} \tilde{\varphi}_1 \, \mathrm{d}X = h + \frac{C}{2}$$

- Method 1: incorporate Dirichlet values through a $B(x)$ function and demand $\psi_i = 0$ where Dirichlet values apply
- Method 2: drop $B(x)$, drop demands to ψ_i, just assemble as if there were no Dirichlet conditions, and modify the linear system instead

Method 2: always choose $\psi_i = \varphi_i$ for all $i \in \mathcal{I}_s$ and set

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x), \quad \mathcal{I}_s = \{0, \ldots, N = N_n\}$$

Attractive way of incorporating Dirichlet conditions

u is treated as unknown at all boundaries when computing entries in the linear system

$$-u'' = 2, \quad u(0) = 0, \ u(L) = D$$

Assemble as if there were no Dirichlet conditions:

$$\frac{1}{h}
\begin{pmatrix}
1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
-1 & 2 & -1 & \ddots & & & & & \vdots \\
0 & -1 & 2 & -1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
& & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
& & & 0 & -1 & 2 & -1 & \ddots & \vdots \\
& & & & \ddots & \ddots & \ddots & \ddots & 0 \\
& & & & & \ddots & \ddots & \ddots & -1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1
\end{pmatrix}
\begin{pmatrix}
c_0 \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
c_N
\end{pmatrix}
=
\begin{pmatrix}
h \\
2h \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
2h \\
h
\end{pmatrix}$$

- Dirichlet condition $u(x_k) = U_k$ means $c_k = U_k$ (since $c_k = u(x_k)$)
- Replace first row by $c_0 = 0$
- Replace last row by $c_N = D$

$$\frac{1}{h}\begin{pmatrix} h & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix}$$

In cell 0 we know u for local node (degree of freedom) $r = 0$. Replace the first cell equation by $\tilde{c}_0 = 0$:

$$\tilde{A}^{(0)} = A = \frac{1}{h} \begin{pmatrix} h & 0 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} 0 \\ h \end{pmatrix}$$

In cell N_e we know u for local node $r = 1$. Replace the last equation in the cell system by $\tilde{c}_1 = D$:

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix}$$

- The modification above destroys symmetry of the matrix: e.g., $A_{0,1} \neq A_{1,0}$
- Symmetry is often important in 2D and 3D (faster computations, less storage)
- A more complex modification can preserve symmetry!

Algorithm for incorporating $c_i = U_i$ in a symmetric way:

1. Subtract column i times U_i from the right-hand side
2. Zero out column and row no i
3. Place 1 on the diagonal
4. Set $b_i = U_i$

$$\frac{1}{h}\begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h + \frac{D}{h} \\ D \end{pmatrix}$$

Symmetric modification applied to $\tilde{A}^{(N_e)}$:

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \tilde{b}^{(N-1)} = \begin{pmatrix} h + D/h \\ D \end{pmatrix}$$

Neumann conditions

How can we incorporate $u'(0) = C$ with finite elements?

$$-u'' = f, \quad u'(0) = C, \; u(L) = D$$

- $\psi_i(L) = 0$ because of Dirichlet condition $u(L) = D$ (or no demand and modify linear system)
- No demand to $\psi_i(0)$
- The condition $u'(0) = C$ will be handled (as usual) through a boundary term arising from integration by parts

Galerkin's method:

$$\int_0^L (u''(x) + f(x))\psi_i(x)dx = 0, \quad i \in \mathcal{I}_s$$

Integration of $u''\psi_i$ by parts:

$$\int_0^L u'(x)\psi_i'(x)\,\mathrm{d}x - (u'(L)\psi_i(L) - u'(0)\psi_i(0)) - \int_0^L f(x)\psi_i(x)\,\mathrm{d}x = 0$$

- $u'(L)\psi_i(L) = 0$ since $\psi_i(L) = 0$
- $u'(0)\psi_i(0) = C\psi_i(0)$ since $u'(0) = C$

Method 1: Boundary function and exclusion of Dirichlet degrees of freedom

- $\psi_i = \varphi_i$, $i \in \mathcal{I}_s = \{0, \ldots, N = N_n - 1\}$
- $B(x) = D\varphi_{N_n}(x)$, $u = B + \sum_{j=0}^{N} c_j \varphi_j$

$$\int_0^L u'(x)\varphi_i'(x)dx = \int_0^L f(x)\varphi_i(x)dx - C\varphi_i(0), \quad i \in \mathcal{I}_s$$

$$\sum_{j=0}^{N} \left(\int_0^L \varphi_i'\varphi_j' dx \right) c_j = \int_0^L \left(f\varphi_i - D\varphi_N'\varphi_i \right) dx - C\varphi_i(0)$$

for $i = 0, \ldots, N = N_n - 1$.

Method 2: Use all φ_i and insert the Dirichlet condition in the linear system

- Now $\psi_i = \varphi_i$, $i = 0, \ldots, N = N_n$
- $\varphi_N(L) \neq 0$, so $u'(L)\varphi_N(L) \neq 0$
- However, the term $u'(L)\varphi_N(L)$ in b_N *will be erased* when we insert the Dirichlet value in $b_N = D$

We can therefore forget about the term $u'(L)\varphi_i(L)$!

Boundary terms $u'\varphi_i$ at points x_i where Dirichlet values apply can always be forgotten.

$$u(x) = \sum_{j=0}^{N=N_n} c_j \varphi_j(x)$$

$$\sum_{j=0}^{N=N_n} \left(\int_0^L \varphi_i'(x)\varphi_j'(x)dx \right) c_j = \int_0^L f(x)\varphi_i(x)dx - C\varphi_i(0)$$

The extra term $C\varphi_0(0)$ affects only the element vector from the first cell since $\varphi_0 = 0$ on all other cells.

$$\tilde{A}^{(0)} = A = \frac{1}{h}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} h - C \\ h \end{pmatrix}$$

The differential equation problem defines the integrals in the variational formulation.

Request these functions from the user:

```
integrand_lhs(phi, r, s, x)
boundary_lhs(phi, r, s, x)
integrand_rhs(phi, r, x)
boundary_rhs(phi, r, x)
```

Must also have a mesh with `vertices`, `cells`, and `dof_map`

Python pseudo code; the element matrix and vector

```python
<Declare global matrix, global rhs: A, b>

# Loop over all cells
for e in range(len(cells)):

    # Compute element matrix and vector
    n = len(dof_map[e])   # no of dofs in this element
    h = vertices[cells[e][1]] - vertices[cells[e][0]]
    <Declare element matrix, element vector: A_e, b_e>

    # Integrate over the reference cell
    points, weights = <numerical integration rule>
    for X, w in zip(points, weights):
        phi = <basis functions + derivatives at X>
        detJ = h/2
        x = <affine mapping from X>
        for r in range(n):
            for s in range(n):
                A_e[r,s] += integrand_lhs(phi, r, s, x)*detJ*w
            b_e[r] += integrand_rhs(phi, r, x)*detJ*w

    # Add boundary terms
    for r in range(n):
        for s in range(n):
            A_e[r,s] += boundary_lhs(phi, r, s, x)*detJ*w
        b_e[r] += boundary_rhs(phi, r, x)*detJ*w
```

```python
for e in range(len(cells)):
    ...

    # Incorporate essential boundary conditions
    for r in range(n):
        global_dof = dof_map[e][r]
        if global_dof in essbc_dofs:
            # dof r is subject to an essential condition
            value = essbc_docs[global_dof]
            # Symmetric modification
            b_e -= value*A_e[:,r]
            A_e[r,:] = 0
            A_e[:,r] = 0
            A_e[r,r] = 1
            b_e[r] = value

    # Assemble
    for r in range(n):
        for s in range(n):
            A[dof_map[e][r], dof_map[e][r]] += A_e[r,s]
        b[dof_map[e][r]] += b_e[r]

<solve linear system>
```

How to do integration by parts is the major difference when moving to 2D and 3D.

Rule for multi-dimensional integration by parts

$$-\int_{\Omega} \nabla \cdot (a(\boldsymbol{x})\nabla u)v \, \mathrm{d}x = \int_{\Omega} a(\boldsymbol{x})\nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial\Omega} a\frac{\partial u}{\partial n} v \, \mathrm{d}s$$

- $\int_{\Omega}() \, \mathrm{d}x$: area (2D) or volume (3D) integral
- $\int_{\partial\Omega}() \, \mathrm{d}s$: line(2D) or surface (3D) integral

- $\partial\Omega_N$: Neumann conditions $-a\frac{\partial u}{\partial n} = g$
- $\partial\Omega_D$: Dirichlet conditions $u = u_0$
- $v \in V$ must vanish on $\partial\Omega_D$ (in method 1)

$$\boldsymbol{v} \cdot \nabla u + \alpha u = \nabla \cdot (a \nabla u) + f, \qquad \boldsymbol{x} \in \Omega$$

$$u = u_0, \qquad \boldsymbol{x} \in \partial \Omega_D$$

$$-a \frac{\partial u}{\partial n} = g, \qquad \boldsymbol{x} \in \partial \Omega_N$$

- Known: a, α, f, u_0, and g.
- Second-order PDE: must have *exactly one boundary condition at each point of the boundary*

Method 1 with boundary function and $\psi_i = 0$ on $\partial \Omega_D$:

$$u(\boldsymbol{x}) = B(\boldsymbol{x}) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(\boldsymbol{x}), \quad B(\boldsymbol{x}) = u_0(\boldsymbol{x})$$

Galerkin's method: multiply by $v \in V$ and integrate over Ω,

$$\int_\Omega (\boldsymbol{v} \cdot \nabla u + \alpha u)v \, \mathrm{d}x = \int_\Omega \nabla \cdot (a\nabla u) \, \mathrm{d}x + \int_\Omega fv \, \mathrm{d}x$$

Integrate second-order term by parts:

$$\int_\Omega \nabla \cdot (a\nabla u) \, v \, \mathrm{d}x = -\int_\Omega a\nabla u \cdot \nabla v \, \mathrm{d}x + \int_{\partial\Omega} a\frac{\partial u}{\partial n} v \, \mathrm{d}s,$$

Resulting variational form:

$$\int_\Omega (\boldsymbol{v} \cdot \nabla u + \alpha u)v \, \mathrm{d}x = -\int_\Omega a\nabla u \cdot \nabla v \, \mathrm{d}x + \int_{\partial\Omega} a\frac{\partial u}{\partial n} v \, \mathrm{d}s + \int_\Omega fv \, \mathrm{d}x$$

Note: $v \neq 0$ only on $\partial \Omega_N$:

$$\int_{\partial \Omega} a \frac{\partial u}{\partial n} v \, \mathrm{d}s = \int_{\partial \Omega_N} a \underbrace{\frac{\partial u}{\partial n}}_{-g} v \, \mathrm{d}s = -\int_{\partial \Omega_N} gv \, \mathrm{d}s$$

The final variational form:

$$\int_{\Omega} (\boldsymbol{v} \cdot \nabla u + \alpha u) v \, \mathrm{d}x = -\int_{\Omega} a \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial \Omega_N} gv \, \mathrm{d}s + \int_{\Omega} fv \, \mathrm{d}x$$

Or with inner product notation:

$$(\boldsymbol{v} \cdot \nabla u, v) + (\alpha u, v) = -(a \nabla u, \nabla v) - (g, v)_N + (f, v)$$

$(g, v)_N$: line or surface integral over $\partial \Omega_N$.

$$u = B + \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad B = u_0$$

$$A_{i,j} = (\boldsymbol{v} \cdot \nabla \psi_j, \psi_i) + (\alpha \psi_j, \psi_i) + (a \nabla \psi_j, \nabla \psi_i)$$

$$b_i = (g, \psi_i)_N + (f, \psi_i) - (\boldsymbol{v} \cdot \nabla u_0, \psi_i) + (\alpha u_0, \psi_i) + (a \nabla u_0, \nabla \psi_i)$$

We want to compute an integral in the physical domain by integrating over the reference cell.

$$\int_{\Omega^{(e)}} a(\boldsymbol{x}) \nabla \varphi_i \cdot \nabla \varphi_j \, \mathrm{d}x$$

Mapping from reference to physical coordinates:

$$x(\boldsymbol{X})$$

with Jacobian J,

$$J_{i,j} = \frac{\partial x_j}{\partial X_i}$$

- $\mathrm{d}x \to \det J \, \mathrm{d}X$.
- Must express $\nabla \varphi_i$ by an expression with $\tilde{\varphi}_r$, $i = q(e, r)$: $\nabla \tilde{\varphi}_r(\boldsymbol{X})$
- We want $\nabla_{\boldsymbol{x}} \tilde{\varphi}_r(\boldsymbol{X})$ (derivatives wrt $\boldsymbol{x}$)
- What we readily have is $\nabla_{\boldsymbol{X}} \tilde{\varphi}_r(\boldsymbol{X})$ (derivative wrt $\boldsymbol{X}$)
- Need to transform $\nabla_{\boldsymbol{X}} \tilde{\varphi}_r(\boldsymbol{X})$ to $\nabla_{\boldsymbol{x}} \tilde{\varphi}_r(\boldsymbol{X})$

Can derive

$$\nabla_{\boldsymbol{X}}\tilde{\varphi}_r = J \cdot \nabla_{\boldsymbol{x}}\varphi_i$$
$$\nabla_{\boldsymbol{x}}\varphi_i = \nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X}) = J^{-1} \cdot \nabla_{\boldsymbol{X}}\tilde{\varphi}_r(\boldsymbol{X})$$

Integral transformation from physical to reference coordinates:

$$\int_{\Omega^{(e)}} a(\boldsymbol{x})\nabla_{\boldsymbol{x}}\varphi_i \cdot \nabla_{\boldsymbol{x}}\varphi_j \, \mathrm{d}x = \int_{\tilde{\Omega}^r} a(\boldsymbol{x}(\boldsymbol{X}))(J^{-1}\cdot\nabla_{\boldsymbol{X}}\tilde{\varphi}_r)\cdot(J^{-1}\cdot\nabla\tilde{\varphi}_s)\det J \, \mathrm{d}X$$

Numerical integration

Numerical integration over reference cell triangles and tetrahedra:

$$\int_{\tilde{\Omega}^r} g \, dX = \sum_{j=0}^{n-1} w_j g(\bar{\boldsymbol{X}}_j)$$

Module `numint.py` contains different rules:

```
>>> import numint
>>> x, w = numint.quadrature_for_triangles(num_points=3)
>>> x
[(0.16666666666666666,  0.16666666666666666),
 (0.66666666666666666,  0.16666666666666666),
 (0.16666666666666666,  0.66666666666666666)]
>>> w
[0.16666666666666666,  0.16666666666666666,  0.16666666666666666]
```

- Triangle: rules with $n = 1, 3, 4, 7$ integrate exactly polynomials of degree $1, 2, 3, 4$, resp.
- Tetrahedron: rules with $n = 1, 4, 5, 11$ integrate exactly polynomials of degree $1, 2, 3, 4$, resp.

- So far: used the finite element framework for discretizing in space
- What about $u_t = u_{xx} + f$?
  1. Use finite differences in time to obtain a set of recursive spatial problems
  2. Solve the spatial problems by the finite element method

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u + f(\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \Omega, t \in (0, T]$$

$$u(\boldsymbol{x}, 0) = I(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega$$

$$\frac{\partial u}{\partial n} = 0, \qquad \boldsymbol{x} \in \partial\Omega, \ t \in (0, T]$$

$$[D_t^+ u = \alpha \nabla^2 u + f]^n, \quad n = 1, 2, \ldots, N_t - 1$$

Solving wrt u^{n+1}:

$$u^{n+1} = u^n + \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right)$$

- $u^n = \sum_j c_j^n \psi_j \in V$, $u^{n+1} = \sum_j c_j^{n+1} \psi_j \in V$
- Compute u^0 from I
- Compute u^{n+1} from u^n by solving the PDE for u^{n+1} at each time level

- $u_e(\boldsymbol{x}, t)$: exact solution of the PDE problem
- $u_e^n(\boldsymbol{x})$: exact solution of time-discrete problem (after applying a finite difference scheme in time)
- $u_e^n(\boldsymbol{x}) \approx u^n = \sum_{j \in \mathcal{I}_s} c_j^n \psi_j =$ solution of the time- and space-discrete problem (after applying a Galerkin method in space)

$$\frac{\partial u_e}{\partial t} = \alpha \nabla^2 u_e + f(\boldsymbol{x}, t)$$

$$u_e^{n+1} = u_e^n + \Delta t \left(\alpha \nabla^2 u_e^n + f(\boldsymbol{x}, t_n) \right)$$

$$u_e^n \approx u^n = \sum_{j=0}^{N} c_j^n \psi_j(\boldsymbol{x}), \quad u_e^{n+1} \approx u^{n+1} = \sum_{j=0}^{N} c_j^{n+1} \psi_j(\boldsymbol{x})$$

$$R = u^{n+1} - u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right)$$

A Forward Euler scheme; weighted residual (or Galerkin) principle

$$R = u^{n+1} - u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right)$$

The weighted residual principle:

$$\int_\Omega R w \, \mathrm{d}x = 0, \quad \forall w \in W$$

results in

$$\int_\Omega \left[u^{n+1} - u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right) \right] w \, \mathrm{d}x = 0, \quad \forall w \in W$$

Galerkin: $W = V$, $w = v$

Isolating the unknown u^{n+1} on the left-hand side:

$$\int_\Omega u^{n+1}\psi_i \, dx = \int_\Omega \left[u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right) \right] v \, dx$$

Integration by parts of $\int \alpha(\nabla^2 u^n) v \, dx$:

$$\int_\Omega \alpha(\nabla^2 u^n) v \, dx = -\int_\Omega \alpha \nabla u^n \cdot \nabla v \, dx + \underbrace{\int_{\partial\Omega} \alpha \frac{\partial u^n}{\partial n} v \, dx}_{=0 \quad \Leftarrow \quad \partial u^n/\partial n = 0}$$

Variational form:

$$\int_\Omega u^{n+1} v \, dx = \int_\Omega u^n v \, dx - \Delta t \int_\Omega \alpha \nabla u^n \cdot \nabla v \, dx + \Delta t \int_\Omega f^n v \, dx, \quad \forall v \in V$$

- u and u: the spatial unknown function to be computed
- u_1 and u_1: the spatial function at the previous time level $t - \Delta t$
- u_2 and u_2: the spatial function at $t - 2\Delta t$
- This new notation gives close correspondence between code and math

$$\int_\Omega uv \, dx = \int_\Omega u_1 v \, dx - \Delta t \int_\Omega \alpha \nabla u_1 \cdot \nabla v \, dx + \Delta t \int_\Omega f^n v \, dx$$

or shorter

$$(u, \psi_i) = (u_1, v) - \Delta t(\alpha \nabla u_1, \nabla v) + (f^n, v)$$

- $u = \sum_{j=0}^{N} c_j \psi_j(\boldsymbol{x})$
- $u_1 = \sum_{j=0}^{N} c_{1,j} \psi_j(\boldsymbol{x})$
- $\forall v \in V$: for $v = \psi_i$, $i = 0, \ldots, N$

Insert these in

$$(u, \psi_i) = (u_1, \psi_i) - \Delta t(\alpha \nabla u_1, \nabla \psi_i) + (f^n, \psi_i)$$

and order terms as matrix-vector products ($i = 0, \ldots, N$):

$$\sum_{j=0}^{N} \underbrace{(\psi_i, \psi_j)}_{M_{i,j}} c_j = \sum_{j=0}^{N} \underbrace{(\psi_i, \psi_j)}_{M_{i,j}} c_{1,j} - \Delta t \sum_{j=0}^{N} \underbrace{(\nabla \psi_i, \alpha \nabla \psi_j)}_{K_{i,j}} c_{1,j} + (f^n, \psi_i)$$

$$Mc = Mc_1 - \Delta t K c_1 + f$$

$$M = \{M_{i,j}\}, \quad M_{i,j} = (\psi_i, \psi_j), \quad i, j \in \mathcal{I}_s$$
$$K = \{K_{i,j}\}, \quad K_{i,j} = (\nabla \psi_i, \alpha \nabla \psi_j), \quad i, j \in \mathcal{I}_s$$
$$f = \{(f(\boldsymbol{x}, t_n), \psi_i)\}_{i \in \mathcal{I}_s}$$
$$c = \{c_i\}_{i \in \mathcal{I}_s}$$
$$c_1 = \{c_{1,i}\}_{i \in \mathcal{I}_s}$$

1. Compute M and K.
2. Initialize u^0 by either interpolation or projection
3. For $n = 1, 2, \ldots, N_t$:
   1. compute $b = Mc_1 - \Delta t K c_1 + f$
   2. solve $Mc = b$
   3. set $c_1 = c$

Initial condition:

- Either interpolation: $c_{1,j} = I(\boldsymbol{x}_j)$ (finite elements)
- Or projection: solve $\sum_j M_{i,j} c_{1,j} = (I, \psi_i)$, $i \in \mathcal{I}_s$

- P1 elements in 1D
- Uniform mesh on $[0, L]$ with cell length h
- No Dirichlet conditions: $\psi_i = \varphi_i$, $i = 0, \ldots, N = N_n$
- Have found formulas for M and K at the element level
- Have assembled the global matrices
- Have developed corresponding finite difference operator formulas
- M: $h[D_t^+(u + \frac{1}{6}h^2 D_x D_x u)]_i^n$
- K: $h[\alpha D_x D_x u]_i^n$

Diffusion equation with finite elements is equivalent to

$$[D_t^+(u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n$$

Can lump the mass matrix by Trapezoidal integration and get the standard finite difference scheme

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n$$

Backward Euler scheme in time:

$$[D_t^- u = \alpha \nabla^2 u + f(\boldsymbol{x}, t)]^n$$

$$u_e^n - \Delta t \left(\alpha \nabla^2 u_e^n + f(\boldsymbol{x}, t_n) \right) = u_e^{n-1}$$

$$u_e^n \approx u^n = \sum_{j=0}^{N} c_j^n \psi_j(\boldsymbol{x}), \quad u_e^{n+1} \approx u^{n+1} = \sum_{j=0}^{N} c_j^{n+1} \psi_j(\boldsymbol{x})$$

$$\int_\Omega \left(u^n v + \Delta t \alpha \nabla u^n \cdot \nabla v\right) \, \mathrm{d}x = \int_\Omega u^{n-1} v \, \mathrm{d}x - \Delta t \int_\Omega f^n v \, \mathrm{d}x, \quad \forall v \in V$$

or

$$(u, v) + \Delta t (\alpha \nabla u, \nabla v) = (u_1, v) + \Delta t (f^n, \psi_i)$$

The linear system: insert $u = \sum_j c_j \psi_i$ and $u_1 = \sum_j c_{1,j} \psi_i$,

$$(M + \Delta t \alpha K) c = M c_1 + f$$

Can interpret the resulting equation system as

$$[D_t^-(u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n$$

Lumped mass matrix (by Trapezoidal integration) gives a standard finite difference method:

$$[D_t^- u = \alpha D_x D_x u + f]_i^n$$

Dirichlet condition at $x = 0$ and Neumann condition at $x = L$:

$$u(\boldsymbol{x}, t) = u_0(\boldsymbol{x}, t), \qquad\qquad \boldsymbol{x} \in \partial\Omega_D$$

$$-\alpha\frac{\partial}{\partial n}u(\boldsymbol{x}, t) = g(\boldsymbol{x}, t), \qquad\qquad \boldsymbol{x} \in \partial\Omega_N$$

Forward Euler in time, Galerkin's method, and integration by parts:

$$\int_\Omega u^{n+1}v\,\mathrm{d}x = \int_\Omega (u^n - \Delta t\alpha\nabla u^n \cdot \nabla v)\,\mathrm{d}x - \Delta t\int_{\partial\Omega_N} gv\,\mathrm{d}s, \quad \forall v \in V$$

Requirement: $v = 0$ on $\partial\Omega_D$

$$u^n(\boldsymbol{x}) = u_0(\boldsymbol{x}, t_n) + \sum_{j \in \mathcal{I}_s} c_j^n \psi_j(\boldsymbol{x})$$

$$\sum_{j \in \mathcal{I}_s} \left(\int_\Omega \psi_i \psi_j \, \mathrm{d}x \right) c_j^{n+1} = \sum_{j \in \mathcal{I}_s} \left(\int_\Omega (\psi_i \psi_j - \Delta t \alpha \nabla \psi_i \cdot \nabla \psi_j) \, \mathrm{d}x \right) c_j^n -$$

$$\int_\Omega \left(u_0(\boldsymbol{x}, t_{n+1}) - u_0(\boldsymbol{x}, t_n) + \Delta t \alpha \nabla u_0(\boldsymbol{x}, t_n) \cdot \nabla \psi_i \right) \mathrm{d}x$$

$$+ \Delta t \int_\Omega f \psi_i \, \mathrm{d}x - \Delta t \int_{\partial \Omega_N} g \psi_i \, \mathrm{d}s, \quad i \in \mathcal{I}_s$$

Finite element basis functions

- $B(x, t_n) = \sum_{j \in I_b} U_j^n \varphi_j$
- $\psi_i = \varphi_{\nu(j)}, \, j \in \mathcal{I}_s$
- $\nu(j), \, j \in \mathcal{I}_s$, are the node numbers corresponding to all nodes without a Dirichlet condition

$$u^n = \sum_{j \in I_b} U_j^n \varphi_j + \sum_{j \in \mathcal{I}_s} c_{1,j} \varphi_{\nu(j)},$$

$$u^{n+1} = \sum_{j \in I_b} U_j^{n+1} \varphi_j + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}$$

$$\sum_{j \in \mathcal{I}_s} \left(\int_\Omega \varphi_i \varphi_j \, \mathrm{d}x \right) c_j = \sum_{j \in \mathcal{I}_s} \left(\int_\Omega (\varphi_i \varphi_j - \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j) \, \mathrm{d}x \right) c_{1,j} -$$

$$\sum_{j \in I_b} \int_\Omega \left(\varphi_i \varphi_j (U_j^{n+1} - U_j^n) + \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j U_j^n \right) \mathrm{d}x$$

$$+ \Delta t \int_\Omega f \varphi_i \, \mathrm{d}x - \Delta t \int_{\partial \Omega_N} g \varphi_i \, \mathrm{d}s, \quad i \in \mathcal{I}_s$$

- Drop boundary function
- Compute as if there are not Dirichlet conditions
- Modify the linear system to incorporate Dirichlet conditions
- $\mathcal{I}_s$ holds the indices of all nodes $\{0, 1, \ldots, N = N_n\}$

$$\sum_{j \in \mathcal{I}_s} \left(\underbrace{\int_\Omega \varphi_i \varphi_j \, \mathrm{d}x}_{M_{i,j}} \right) c_j = \sum_{j \in \mathcal{I}_s} \left(\underbrace{\int_\Omega \varphi_i \varphi_j \, \mathrm{d}x}_{M_{i,j}} - \Delta t \underbrace{\int_\Omega \alpha \nabla \varphi_i \cdot \nabla \varphi_j \, \mathrm{d}x}_{K_{i,j}} \right) c_{1,j}$$

$$\underbrace{- \Delta t \int_\Omega f \varphi_i \, \mathrm{d}x - \Delta t \int_{\partial \Omega_N} g \varphi_i \, \mathrm{d}s}_{f_i}, \quad i \in \mathcal{I}_s$$

$$Mc = b, \quad b = Mc_1 - \Delta t K c_1 + \Delta t f$$

For each k where a Dirichlet condition applies, $u(x_k, t_{n+1}) = U_k^{n+1}$,

- set row k in M to zero and 1 on the diagonal: $M_{k,j} = 0$, $j \in \mathcal{I}_s$, $M_{k,k} = 1$
- $b_k = U_k^{n+1}$

Or apply the slightly more complicated modification which preserves symmetry of M

Backward Euler discretization in time gives a more complicated coefficient matrix:

$$Ac = b, \quad A = M + \Delta t K, \quad b = M c_1 + \Delta t f$$

- Set row k to zero and 1 on the diagonal: $M_{k,j} = 0$, $j \in \mathcal{I}_s$, $M_{k,k} = 1$
- Set row k to zero: $K_{k,j} = 0$, $j \in \mathcal{I}_s$
- $b_k = U_k^{n+1}$

Observe: $A_{k,k} = M_{k,k} + \Delta t K_{k,k} = 1 + 0$, so $c_k = U_k^{n+1}$

The diffusion equation $u_t = \alpha u_{xx}$ allows a (Fourier) wave component

$$u = A_e^n e^{ikx}, \quad A_e = e^{-\alpha k^2 \Delta t}$$

Numerical schemes often allow the similar solution

$$u_q^n = A^n e^{ikx}$$

- A: amplification factor to be computed
- How good is this A compared to the exact one?

$$[D_t^+ A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{A-1}{\Delta t},$$

$$[D_t^- A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{1-A^{-1}}{\Delta t},$$

$$[D_t A^n e^{ikq\Delta x}]^{n+\frac{1}{2}} = A^{n+\frac{1}{2}} e^{ikq\Delta x} \frac{A^{\frac{1}{2}} - A^{-\frac{1}{2}}}{\Delta t} = A^n e^{ikq\Delta x} \frac{A-1}{\Delta t},$$

$$[D_x D_x A^n e^{ikq\Delta x}]_q = -A^n \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right)$$

Introduce $p = k\Delta x/2$ and $C = \alpha\Delta t/\Delta x^2$:

$$A = 1 - 4C\,\frac{\sin^2 p}{\underbrace{1 - \frac{2}{3}\sin^2 p}_{\text{from } M}}$$
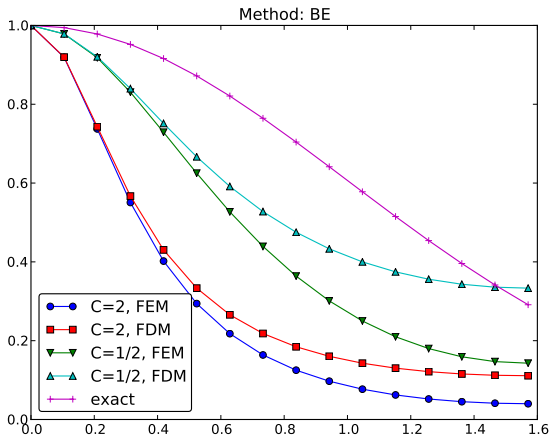
(See notes for details)

Stability: $|A| \leq 1$:

$$C \leq \frac{1}{6} \quad \Rightarrow \quad \Delta t \leq \frac{\Delta x^2}{6\alpha}$$

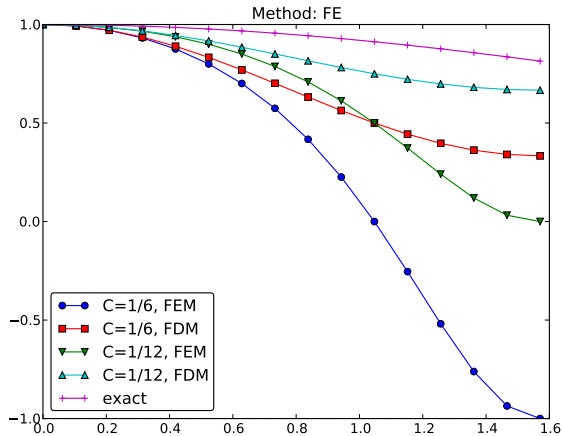Finite differences: $C \leq \frac{1}{2}$, so finite elements give a *stricter* stability criterion for this PDE!

Coarse meshes:

$$A = \left(1 + 4\,C\,\frac{\sin^2 p}{1 + \frac{2}{3}\sin^2 p}\right)^{-1} \quad \text{(unconditionally stable)}$$

Method: BE

Method: BE